

J2EE
WITH
STRUTS FRAMEWORK

Disclaimer

The copyright of the content used in the courseware will remain with Principle Company.

TABLE OF CONTENTS

<u>Ch.No</u>	<u>Chapter Name</u>	<u>Page No.</u>
1.	Introduction to J2EE	3
2.	Using JDBC API	19
3.	Introduction to HTML	53
4.	Web Application & Java Servlet Technology	68
5.	Inside Servlet	78
6.	Anvanced Servlet Part-1	100
7.	Anvanced Servlet Part-2	128
8.	Anvanced Servlet Part-3	139
9.	JSP Basics	176
10.	Advanced JSP	194
11.	Using JSP Custom Tags	225
12.	Working with JSTL	247
13.	MVC Based Web Development with Struts	276

Introduction to J2EE

Developers today increasingly recognize the need for distributed, transactional, and portable applications that leverage the speed, security, and reliability of server-side technology. In the world of information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources.

With the Java Platform, Enterprise Edition (Java EE), development of Java enterprise applications has never been easier or faster. The aim of the Java EE 5 platform is to provide developers a powerful set of APIs while reducing development time, reducing application complexity, and improving application performance.

The Java EE 5 platform introduces a simplified programming model. With Java EE 5 technology, XML deployment descriptors are now optional. Instead, a developer can simply enter the information as an annotation directly into a Java source file, and the Java EE server will configure the component at deployment and runtime. These annotations are generally used to embed in a program data that would otherwise be furnished in a deployment descriptor. With annotations, the specification information is put directly in your code next to the program element that it affects.

In the Java EE platform, dependency injection can be applied to all resources that a component needs, effectively hiding the creation and lookup of resources from application code. Dependency injection can be used in EJB containers, web containers, and application clients. Dependency injection allows the Java EE container to automatically insert references to other required components or resources using annotations.

The Java Persistence API is new to the Java EE 5 platform. The Java Persistence API provides an object/relational mapping for managing relational data in enterprise beans, web components, and application clients. It can also be used in Java SE applications, outside of the Java EE environment.

Java EE Application Model

The Java EE application model begins with the Java programming language and the Java virtual machine. The proven portability, security, and developer productivity they provide forms the basis of the application model. Java EE is designed to support applications that implement enterprise services for customers, employees, suppliers, partners, and others who make demands on or

contributions to the enterprise. Such applications are inherently complex, potentially accessing data from a variety of sources and distributing applications to a variety of clients.

To better control and manage these applications, the business functions to support these various users are conducted in the middle tier. The middle tier represents an environment that is closely controlled by an enterprise's information technology department. The middle tier is typically run on dedicated server hardware and has access to the full services of the enterprise.

The Java EE application model defines an architecture for implementing services as multitier applications that deliver the scalability, accessibility, and manageability needed by enterprise-level applications. This model partitions the work needed to implement a multitier service into two parts: the business and presentation logic to be implemented by the developer, and the standard system services provided by the Java EE platform. The developer can rely on the platform to provide solutions for the hard systems-level problems of developing a multitier service.

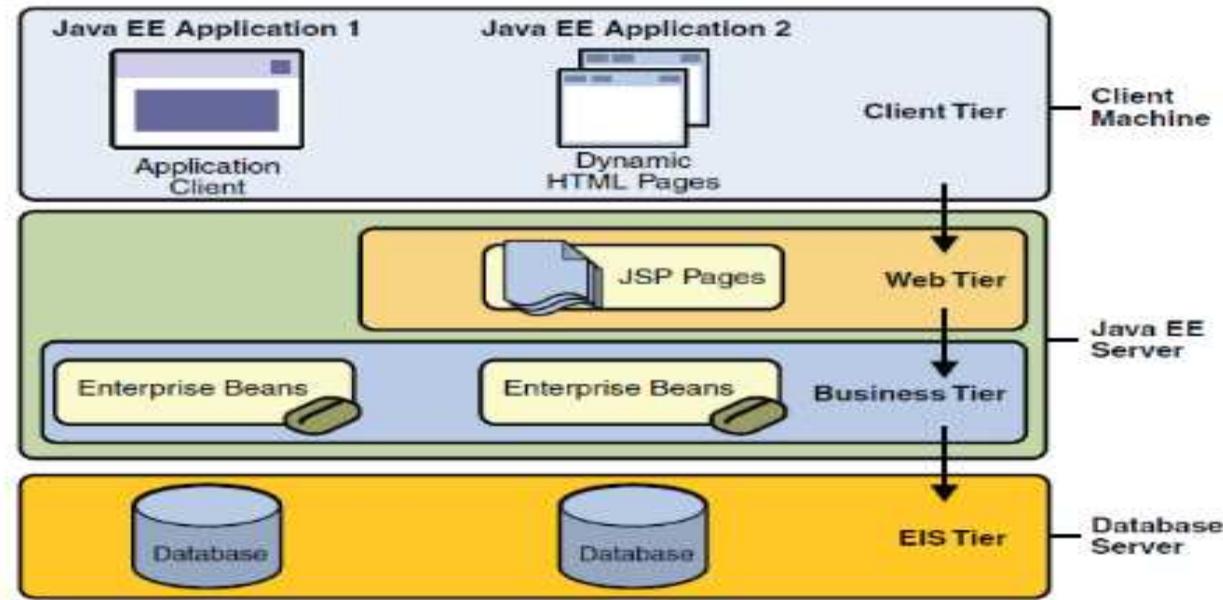
Distributed Multitiered Applications

The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a Java EE application are installed on different machines depending on the tier in the multitiered Java EE environment to which the application component belongs.

Figure given below shows two multitier Java EE applications divided into the tiers described in the following list.

- Client-tier components run on the client machine.
- Web-tier components run on the Java EE server.
- Business-tier components run on the Java EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a Java EE application can consist of the three or four tiers shown in the figure given below, Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Java EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.



Security

While other enterprise application models require platform-specific security measures in each application, the Java EE security environment enables security constraints to be defined at deployment time. The Java EE platform makes applications portable to a wide variety of security implementations by shielding application developers from the complexity of implementing security features.

The Java EE platform provides standard declarative access control rules that are defined by the developer and interpreted when the application is deployed on the server. Java EE also provides standard login mechanisms so application developers do not have to implement these mechanisms in their applications. The same application works in a variety of different security environments without changing the source code.

Java EE Components

Java EE applications are made up of components. A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.

The Java EE specification defines the following Java EE components:

- Application clients and applets are components that run on the client.
- Java Servlet, JavaServer Faces, and JavaServer Pages (JSP) technology components are web components that run on the server.

- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

Java EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between Java EE components and “standard” Java classes is that Java EE components are assembled into a Java EE application, are verified to be well formed and in compliance with the Java EE specification, and are deployed to production, where they are run and managed by the Java EE server.

Java EE Clients

A Java EE client can be a web client or an application client.

Web Clients

A web client consists of two parts:

- (1) Dynamic web pages containing various types of markup language (HTML, XML, and so on, which are generated by web components running in the web tier, and
- (2) A web browser, which renders the pages received from the server.

A web client is sometimes called a thin client. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

Applets

A web page received from the web tier can include an embedded applet. An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

An application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier. Application clients written in languages other than Java can interact with Java EE 5 servers, enabling the Java EE 5 platform to interoperate with legacy systems, clients, and non-Java languages.

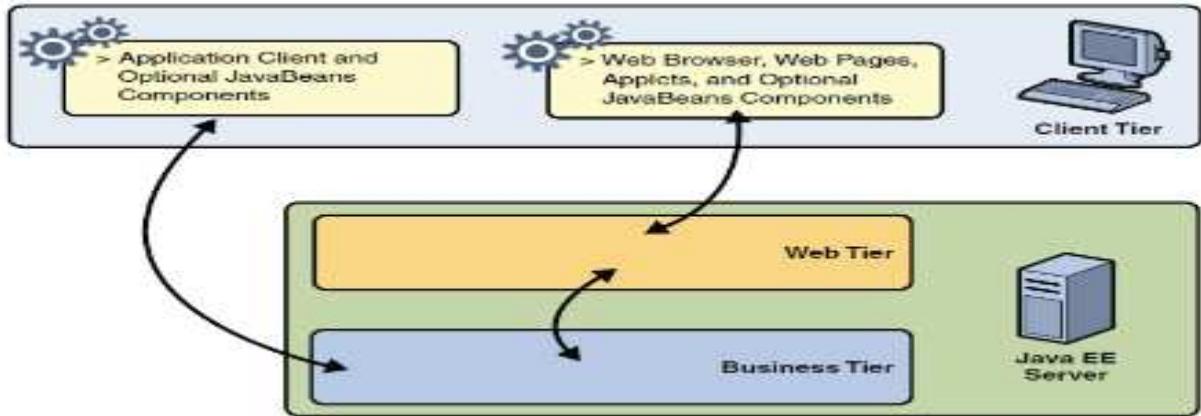
The JavaBeans Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the Java EE server, or between server components and a database. JavaBeans components are not considered Java EE components by the Java EE specification.

JavaBeans components have properties and have get and set methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

Java EE Server Communications

Following figure shows the various elements that can make up the client tier. The client communicates with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the web tier.



Your Java EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy,

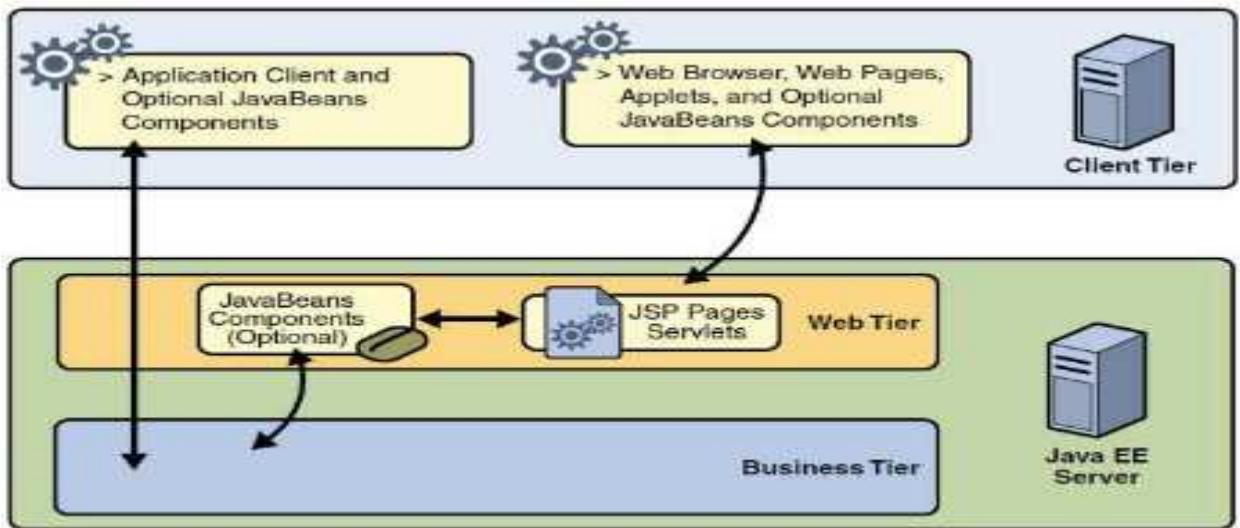
and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.

Web Components

Java EE web components are either servlets or pages created using JSP technology (JSP pages) and/or JavaServer Faces technology. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. JavaServer Faces technology builds on servlets and JSP technology and provides a user interface component framework for web applications.

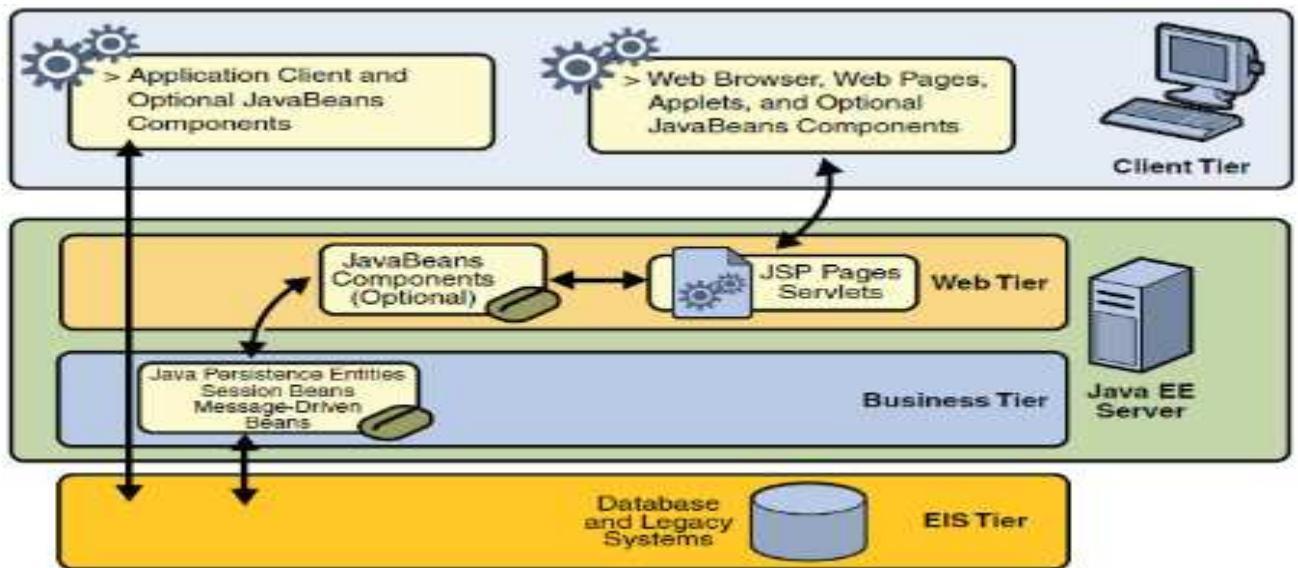
Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the Java EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in the following figure, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.



Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Following figure shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.



Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, Java EE application components might need access to enterprise information systems for database connectivity.

Java EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent Java EE architecture makes Java EE applications easy to write because business logic is organized into reusable components.

In addition, the Java EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, it must be assembled into a Java EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the Java EE application and for the Java EE application itself. Container settings customize the underlying support provided by the Java EE server, including services such as security, transaction management, Java Naming and Directory Interface (JNDI) lookups, and remote connectivity.

Here are some of the highlights:

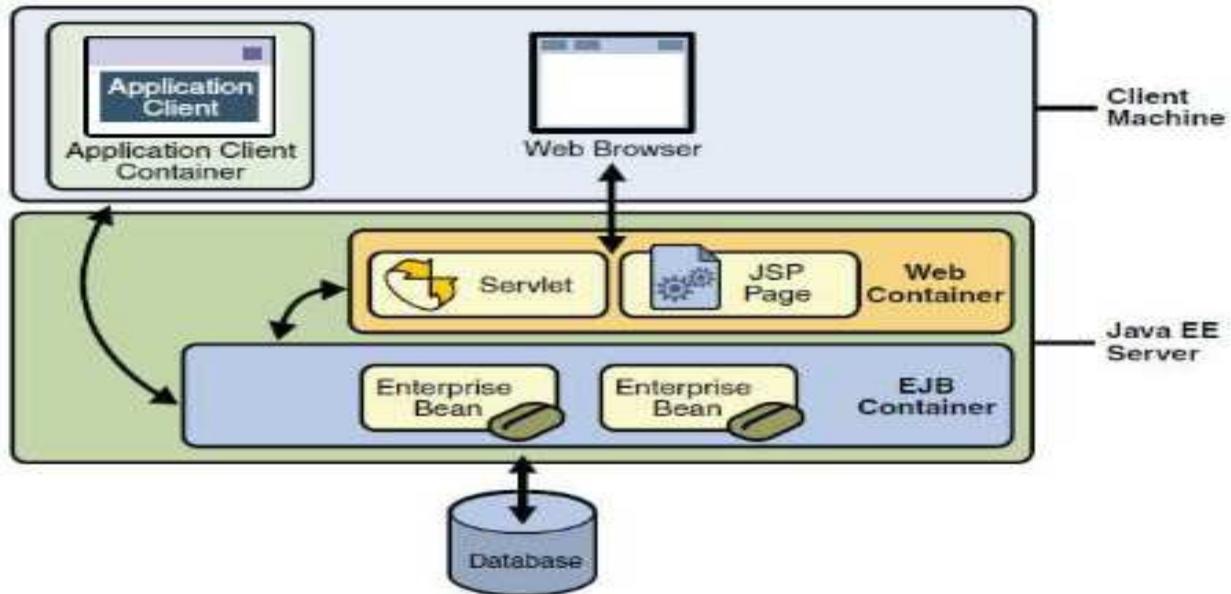
- The Java EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
- The Java EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the Java EE architecture provides configurable services, application components within the same Java EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages non-configurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs.

Container Types

The deployment process installs Java EE application components in the Java EE containers as illustrated in the following figure.



- **Java EE server:** The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.
- **Enterprise JavaBeans (EJB) container:** Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
- **Web container:** Manages the execution of JSP page and servlet components for Java EE applications. Web components and their container run on the Java EE server.
- **Application client container:** Manages the execution of application client components. Application clients and their container run on the client.
- **Applet container:** Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

Java EE Application Assembly and Deployment

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system. Each unit contains:

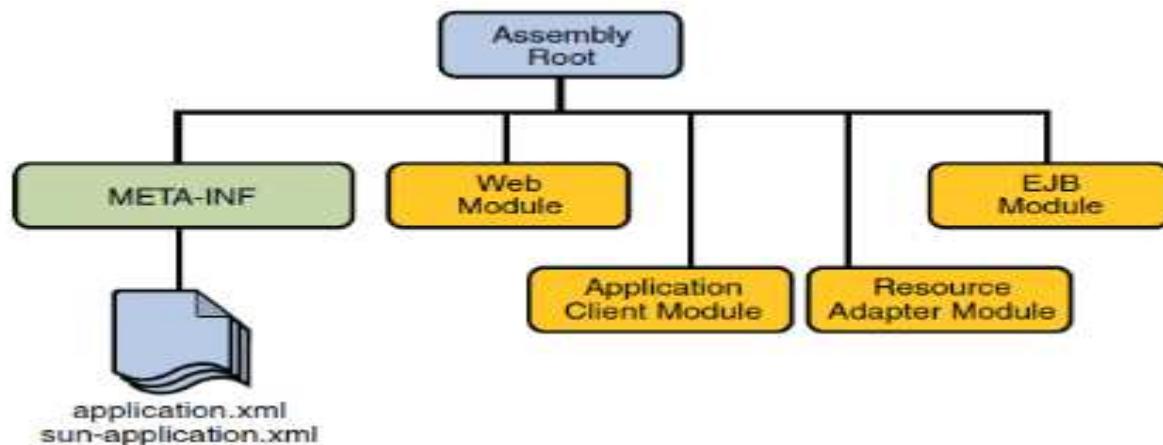
- A functional component or components (such as an enterprise bean, JSP page, servlet, or applet)
- An optional deployment descriptor that describes its content

Once a Java EE unit has been produced, it is ready to be deployed. Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users that can access it and the name of the local database. Once deployed on a local platform, the application is ready to run.

Packaging Applications

A Java EE application is delivered in an Enterprise Archive (EAR) file, a standard Java Archive (JAR) file with an .ear extension. Using EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE EAR files.

An EAR file, as the following figure shows, contains Java EE modules and deployment descriptors. A deployment descriptor is an XML document with an .xml extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.



There are two types of deployment descriptors: Java EE and runtime.

A Java EE deployment descriptor is defined by a Java EE specification and can be used to configure deployment settings on any Java EE-compliant implementation.

A runtime deployment descriptor is used to configure Java EE implementation-specific parameters. For example, the Sun Java System Application Server Platform Edition 9 runtime deployment descriptor contains information such as the context root of a web application, the mapping of portable names of an application's resources to the server's resources, and Application Server implementation-specific parameters, such as caching directives. The Application Server runtime deployment descriptors are named sun-moduleType.xml and are located in the same META-INF directory as the Java EE deployment descriptor.

A Java EE module consists of one or more Java EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise

bean. A Java EE module without an application deployment descriptor can be deployed as a stand-alone module.

The four types of Java EE modules are as follows:

- **EJB modules**, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a .jar extension.
- **Web modules**, which contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a .war (Web ARchive) extension.
- **Application client modules**, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a .jar extension.
- **Resource adapter modules**, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture for a particular EIS. Resource adapter modules are packaged as JAR files with an .rar (resource adapter archive) extension.

Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles involve purchasing and installing the Java EE product and tools. After software is purchased and installed, Java EE components can be developed by application component providers, assembled by application assemblers, and deployed by application deployers. In a large organization, each of these roles might be executed by different individuals or teams. This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in the application component development phase, an enterprise bean software developer delivers EJB JAR files. In the application assembly role, another developer combines these EJB JAR files into a Java EE application and saves it in an EAR file. In the application deployment role, a system administrator at the customer site uses the EAR file to install the Java EE application into a Java EE server.

The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

Java EE Product Provider

The Java EE product provider is the company that designs and makes available for purchase the Java EE platform APIs, and other features defined in the Java EE specification. Product providers are typically application server vendors who implement the Java EE platform according to the Java EE 5 Platform specification.

Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

Application Component Provider

The application component provider is the company or person who creates web components, enterprise beans, applets, or application clients for use in Java EE applications.

Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains one or more enterprise beans:

- Writes and compiles the source code
- Specifies the deployment descriptor
- Packages the .class files and deployment descriptor into the EJB JAR file

Web Component Developer

A web component developer performs the following tasks to deliver a WAR file containing one or more web components:

- Writes and compiles servlet source code
- Writes JSP, JavaServer Faces, and HTML files
- Specifies the deployment descriptor
- Packages the .class, .jsp, and.html files and deployment descriptor into the WAR file

Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client
- Packages the .class files and deployment descriptor into the JAR file

Application Assembler

The application assembler is the company or person who receives application modules from component providers and assembles them into a Java EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or can use tools that correctly add XML tags according to interactive selections.

A software developer performs the following tasks to deliver an EAR file containing the Java EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a Java EE application (EAR) file
- Specifies the deployment descriptor for the Java EE application
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification

Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the Java EE application, administers the computing and networking infrastructure where Java EE applications run, and oversees the runtime environment. Duties include such things as setting transaction controls and security attributes and specifying connections to databases.

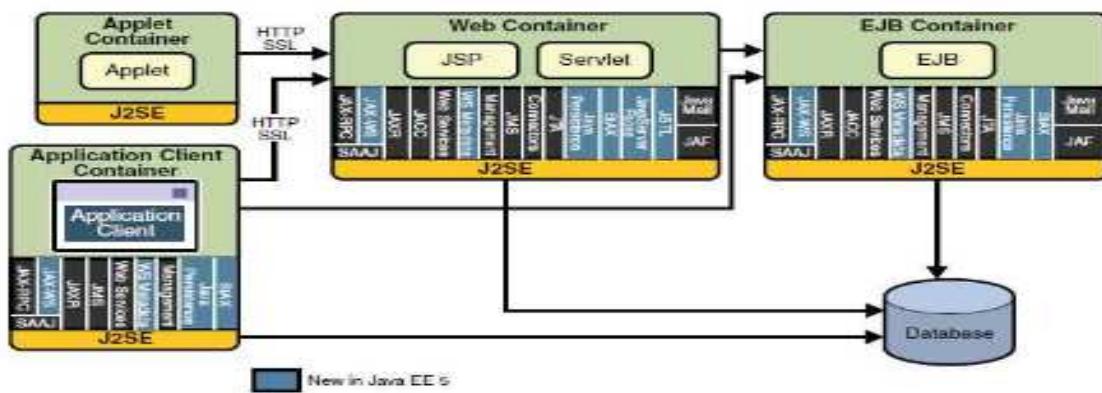
During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer or system administrator performs the following tasks to install and configure a Java EE application:

- Adds the Java EE application (EAR) file created in the preceding phase to the Java EE server
- Configures the Java EE application for the operational environment by modifying the deployment descriptor of the Java EE application
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification
- Deploys (installs) the Java EE application EAR file into the Java EE server

Java EE 5 APIs

The following figure illustrates the availability of the Java EE 5 platform APIs in each Java EE container type.



Enterprise JavaBeans Technology

An Enterprise JavaBeans (EJB) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.

There are two kinds of enterprise beans: session beans and message-driven beans. A session bean represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. A message-driven bean combines features of a session bean and a message listener, allowing a business component to receive messages asynchronously. Commonly, these are Java Message Service (JMS) messages.

In Java EE 5, entity beans have been replaced by Java persistence API entities. An entity represents persistent data stored in one row of a database table. If the client terminates, or if the server shuts down, the persistence manager ensures that the entity data is saved.

Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

Java Server Pages Technology

Java Server Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

Java Server Pages Standard Tag Library

The Java Server Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

Java Server Faces

Java Server Faces technology is a user interface framework for building web applications. The main components of Java Server Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.
- A standard RenderKit for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available using standard Java APIs and XML-based configuration files.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. An auto commit means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

Java Mail API

Java EE applications use the JavaMail API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The Java EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

Java Database Connectivity API

The Java Database Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the Java EE platform.

Java Persistence API

The Java Persistence API is a Java standards-based solution for persistence. Persistence uses an object-relational mapping approach to bridge the gap between an object oriented model and a relational database. Java Persistence consists of three areas:

- The Java Persistence API
- The query language
- Object/relational mapping metadata

Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS,DNS, and NIS. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a Java EE application can store and retrieve any type of named Java object, allowing Java EE applications to coexist with many legacy applications and systems.

Java EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A naming environment allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI naming context.

A Java EE component can locate its environment naming context using JNDI interfaces. A component can create a javax.naming.InitialContext object and looks up the environment naming context in InitialContext under the name java:comp/env. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A Java EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA UserTransaction objects, are stored in the environment naming context, java:comp/env. The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC DataSource objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext java:comp/env/ejb, and JDBC DataSource references in the subcontext java:comp/env/jdbc.

Using JDBC API

It's hard to find a professional web site today that doesn't have some sort of database connectivity. Webmasters have hooked online frontends to all manner of legacy systems, including package tracking and directory databases, as well as many newer systems like online messaging, storefronts, and search engines. But web-database interaction comes with a price: database-backed web sites can be difficult to develop and can often exact heavy performance penalties. Still, for many web sites, database connectivity is just too useful to let go. More and more, databases are driving the Web.

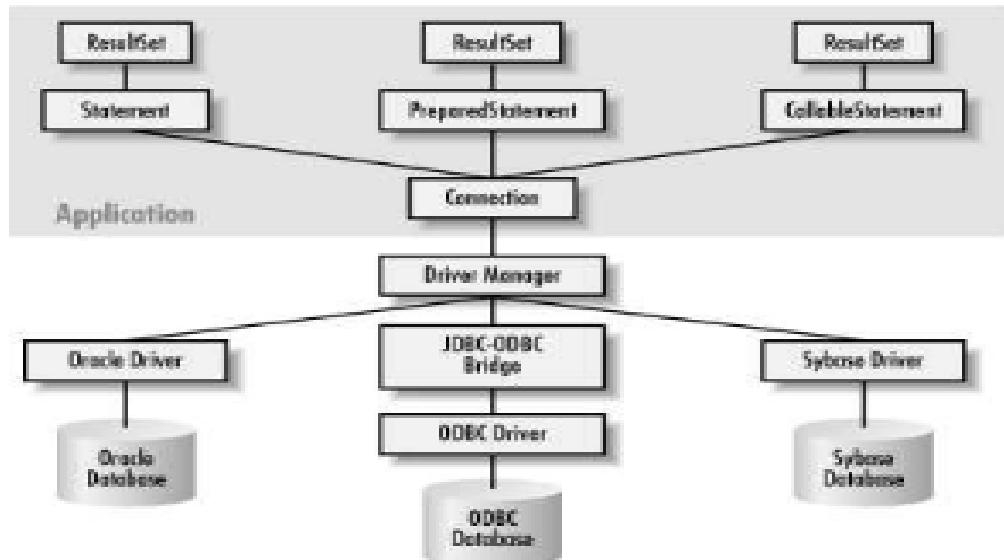


Figure: JAVA Application interacting with Database

JDBC is a SQL-level API—one that allows you to execute SQL statements and retrieve the results, if any. The API itself is a set of interfaces and classes designed to perform actions against any database.

JDBC Drivers

The JDBC API, found in the `java.sql` package, contains only a few concrete classes. Much of the API is distributed as database-neutral interface classes that specify behaviour without providing any implementation. The actual implementations are provided by third-party vendors.

An individual database system is accessed via a specific JDBC driver that implements the `java.sql.Driver` interface. Drivers exist for nearly all popular RDBMS systems, though not all are available for free. Sun bundles a free JDBC-ODBC bridge driver with the JDK to allow access to

standard ODBC data sources, such as a Microsoft Access database. However, Sun advises against using the bridge driver for anything other than development and very limited deployment.

Servlet developers in particular should need this warning because any problem in the JDBC-ODBC bridge driver's native code section can crash the entire server, not just your servlets. JDBC drivers are available for most database platforms, from a number of vendors and in a number of different flavours.

There are four driver categories:

Type 1: JDBC-ODBC Bridge Driver

Type 1 drivers use a bridge technology to connect a Java client to an ODBC database service. Sun's JDBC-ODBC bridge is the most common Type 1 driver. These drivers are implemented using native code.

Type 2: Native-API Partly Java Driver

Type 2 drivers wrap a thin layer of Java around database-specific native code libraries. For Oracle databases, the native code libraries might be based on the OCI (Oracle Call Interface) libraries, which were originally designed for C/C++ programmers. Because Type 2 drivers are implemented using native code, in some cases they have better performance than their all-Java counterparts. They add an element of risk, however, because a defect in a driver's native code section can crash the entire server.

Type 3: Net-Protocol All-Java Driver

Type 3 drivers communicate via a generic network protocol to a piece of custom middleware. The middleware component might use any type of driver to provide the actual database access. These drivers are all Java, which makes them useful for applet deployment and safe for servlet deployment.

Type 4: Native-Protocol All-Java Driver

Type 4 drivers are the most direct of the lot. Written entirely in Java, Type 4 drivers understand database-specific networking protocols and can access the database directly without any additional software.

A list of currently available JDBC drivers can be found at
<http://industry.java.sun.com/products/jdbc/drivers>.

Getting a Connection

The first step in using a JDBC driver to get a database connection involves loading the specific driver class into the application's JVM. This makes the driver available later, when we need it for opening the connection. An easy way to load the driver class is to use the Class.forName() method:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

When the driver is loaded into memory, it registers itself with the java.sql.DriverManager class as an available database driver.

The next step is to ask the DriverManager class to open a connection to a given database, where the database is specified by a specially formatted URL. The method used to open the connection is DriverManager.getConnection(). It returns a class that implements the java.sql.Connection interface:

```
Connection con = DriverManager.getConnection("jdbc:odbc:somedb", "user", "passwd");
```

A JDBC URL identifies an individual database in a driver-specific manner. Different drivers may need different information in the URL to specify the host database. JDBC URLs usually begin with jdbc:subprotocol:subname.

For example, the Oracle JDBC-Thin driver uses a URL of the form of
jdbc:oracle:thin:@dbhost:port:sid ; the JDBC-ODBC bridge uses
jdbc:odbc:datasourcename;odbcoptions.

During the call to *getConnection()*, the DriverManager object asks each registered driver if it recognizes the URL. If a driver says yes, the driver manager uses that driver to create the Connection object. Here is a snippet of code a servlet might use to load its database driver with the JDBC-ODBC bridge and create an initial connection:

```
Connection con = null;
try {
    // Load (and therefore register) the JDBC-ODBC Bridge
    // Might throw a ClassNotFoundException
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Get a connection to the database
    // Might throw an SQLException
    con = DriverManager.getConnection("jdbc:odbc:somedb", "user", "passwd");
    // The rest of the code goes here.
}
catch (ClassNotFoundException e) {
    // Handle an error loading the driver
}
catch (SQLException e) {
    // Handle an error getting the connection
}
finally {
    //Close the Connection to release the database resources immediately.
    try {
        if(con != null) con.close();
    }
    catch (SQLException ignored) { } }
```

There are actually three forms of `getConnection()` available. In addition to the preceding one shown, there's a simpler one that takes just a URL: `getConnection(String url)`. This method can be used when there are no login requirements or when the login information has to be placed in the URL. There's also one form that takes a URL and `Properties` object: `getConnection(String url, Properties props)`. This method provides the most flexibility. The `Properties` object (a `Hashtable` with all keys and values of type `String`) holds the standard user and password properties and in addition may hold any additional properties that might need to be passed to the underlying database driver. For example, some drivers respect the `cacherows` property specifying how many rows to cache at a time. Using this method makes it easy to establish a database connection based on an external `.properties` file. For example, the driver, URL, and credentials to use can be specified in the following `sql.properties` file. This name=value format is standard for Java properties files:

```
connection.driver=sun.jdbc.odbc.JdbcOdbcDriver connection.  
url=jdbc:odbc:somedb  
user=user  
password=passwd
```

The code shown in Example -1 establishes a database connection using the values stored within this `sql.properties` file. Note that the `user` and `password` properties are standard, the `connection.driver` and `connection.url` properties are special names used by the code, and any additional property values will just be passed on to the underlying driver as well.

Example -1. Using a Properties File to Open a Database Connection

```
// Get the properties for the database connection Properties props = new Properties ();  
InputStream in = new FileInputStream("sql.properties"); props.load(in);  
in.close(); // should really be in a finally block  
// Load the driver  
Class.forName(props.getProperty("connection.driver"));  
// Get the connection  
con = DriverManager.getConnection(props.getProperty("connection.url"), props);
```

First the `Properties` object is constructed and filled with the values read from the `sql.properties` file. Then the various properties are used to establish the database connection. Following this approach allows all database connection information to be changed without so much as a recompile of the Java code!

Executing SQL Queries

To really use a database, we need to have some way to execute queries. The simplest way to execute a query is to use the `java.sql.Statement` class. `Statement` objects are never instantiated directly; instead, a program calls the `createStatement()` method of `Connection` to obtain a new `Statement` object:

```
Statement stmt = con.createStatement();
```

A query that returns data can be executed using the executeQuery() method of Statement. This method executes the statement and returns a java.sql.ResultSet that encapsulates the retrieved data:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
```

You can think of a ResultSet object as a representation of the query result returned one row at a time. You use the next() method of ResultSet to move from row to row. The ResultSet interface also boasts a multitude of methods designed for retrieving data from the current row. The getString() and getObject() methods are among the most frequently used for retrieving column values:

```
while(rs.next()) {  
    String event = rs.getString("event");  
    Object count = (Integer) rs.getObject("count"); }
```

You should know that the ResultSet is linked to its parent Statement. Therefore, if a Statement is closed or used to execute another query, any related ResultSet objects are closed automatically.

Example -2 shows a very simple example that uses the Oracle JDBC driver to perform a simple query, printing names and phone numbers for all employees listed in a database table. We assume that the database contains a table named EMPLOYEES, with at least two fields, NAME and PHONE.

Example-2. A JDBC-Enabled Example

```
import java.io.*;  
import java.sql.*;  
  
public class DBPhoneLookup {  
  
    public static void main(String args[]) {  
        Connection con = null;  
        Statement stmt = null;  
        ResultSet rs = null;  
        try {  
            // Load (and therefore register) the Oracle Driver  
            Class.forName("oracle.jdbc.driver.OracleDriver"); // Get a Connection to the database  
            con = DriverManager.getConnection("jdbc:oracle:thin:@dbhost:1528:ORCL", "user", "passwd");  
            // Create a Statement object  
            stmt = con.createStatement();  
            // Execute an SQL query, get a ResultSet  
            rs = stmt.executeQuery("SELECT NAME, PHONE FROM EMPLOYEES");  
            while(rs.next()) {  
                System.out.println(rs.getString("name") + "\t" + rs.getString("phone"));  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

}
catch(ClassNotFoundException e) {
    System.out.println("Couldn't load database driver: " + e.getMessage());
}
catch(SQLException e) {

    out.println("SQLException caught: " + e.getMessage());
}
finally {
    // Always close the database connection.
    try {
        if(con != null) con.close();
    }
    catch (SQLException ignored) { }
}
}
}

```

Handling SQL Exceptions

DBPhoneLookup encloses most of its code in a try/catch block. This block catches two exceptions: ClassNotFoundException and SQLException. The former is thrown by the Class.forName() method when the JDBC driver class cannot be loaded. The latter is thrown by any JDBC method that has a problem. SQLException objects are just like any other exception type, with the additional feature that they can chain. The SQLException class defines an extra method, getNextException(), that allows the exception to encapsulate additional Exception objects. We didn't bother with this feature in the previous example, but here's how to use it:

```

catch (SQLException e) {
    out.println(e.getMessage());
    while((e = e.getNextException()) != null) {
        out.println(e.getMessage());
    }
}

```

This code displays the message from the first exception and then loops through all the remaining exceptions, outputting the error message associated with each one. In practice, the first exception will generally include the most relevant information.

ResultSets in Detail

Before we continue, we should take a closer look at the ResultSet interface and the related ResultSetMetaData interface. In previous example , we knew what our query looked like and we knew what we expected to get back, so we formatted the output appropriately. But, if we want to display the results of a query in an HTML table, it would be nice to have some Java code that builds the table automatically from the ResultSet rather than having to write the same loop-and-display

code over and over. As an added bonus, this kind of code makes it possible to change the contents of the table simply by changing the query.

The ResultSetMetaData interface provides a way for a program to learn about the underlying structure of a query result on the fly. We can use it to build an object that dynamically generates an HTML table from a ResultSet.

```
import java.sql.*;
public class ColumnInfo
{
    public static void main(String args[])
    {
        try
        {
            /*Initialize and load the JDBC-ODBC bridge driver*/
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver"); /*Establish a connection with a data source*/
            Connection con = DriverManager.getConnection ("jdbc:odbc:MyDataSource",
                "administrator","");
            /*Create an SQL statement*/
            String str= "SELECT * FROM "+args[0]+ "";
            Statement stmt = con.createStatement ();
            /*Execute an SQL statement*/
            ResultSet rs = stmt.executeQuery(str);
            ResultSetMetaData rsmd = rs.getMetaData (); rs.next ();
            /*Retrieve the number of columns in a ResultSet*/
            System.out.println ("Number of Attributes in the authors Table:"+rsmd.getColumnCount ());
            System.out.println ("");
            System.out.println ("-----");
            System.out.println ("Attributes of the "+args [0] +" Table");
            System.out.println ("-----");
            /*Retrieve and display the names and data types of various columns in a ResultSet*/
            for (int i = 1; i <= rsmd.getColumnCount (); i++) {
                System.out.println (rsmd.getColumnName(i) + " : " + rsmd.getColumnTypeName(i));
            }
            /*Close the ResultSet object*/
            rs.close();
            /*Close the Statement object*/
            stmt.close();
        }
        catch(Exception e)
        {
            System.out.println("Error : " + e);
        }
    }
}
```

Indexes in ResultSet objects follow the RDBMS standard rather than the C++/Java standard, which means they are numbered from 1 to n rather than from 0 to n-1.

Table -1 shows the Java methods you can use to retrieve some common SQL datatypes from a database. No matter what the type, you can always use the getObject() method of ResultSet, in which case the type of the object returned is shown in the second column. You can also use a specific getXXX() method. These methods are shown in the third column, along with the Java datatypes they return. Remember that supported SQL datatypes vary from database to database.

SQL Data Type	Java Type Returned by getObject()	Recommended Alternative to getObject()
BIGINT	Long	long getLong()
BINARY	byte[]	byte[] getBytes()
BIT	Boolean	boolean getBoolean()
CHAR	String	String getString()
DATE	java.sql.Date	java.sql.Date getDate()
DECIMAL	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
DOUBLE	Double	double getDouble()
FLOAT	Double	double getDouble()
INTEGER	Integer	int getInt()
LONGVARBINARY	byte[]	InputStream getBinaryStream()
LONGVARCHAR	String	InputStream getAsciiStream() InputStream getUnicodeStream()
NUMERIC	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
REAL	Float	float getFloat()
SAMLLINT	Integer	short getByte()
TIME	java.sql.Time	java.sql.Time getTime()
TimeStamp	java.sql.TimeStamp	java.sql.TimeStamp getTimeStamp()
TINYINT	Integer	byte getByte()
VARBINARY	byte[]	byte[] getBytes()
VARCHAR	String	String getString()

Table 1. Methods to retrieve Data from ResultSet

Handling Null Fields

A database field can be set to null to indicate that no value is present, in much the same way that a Java object can be set to null. Handling null database values with JDBC can be a little tricky. A method that doesn't return an object, like getInt(), has no way of indicating whether a column is null or whether it contains actual information. Any special value, like 0, might be a legitimate value. Therefore, JDBC includes the wasNull() method in ResultSet, which returns true or false depending on whether the last column read was a true database null.

This means that you must read data from the ResultSet into a variable, call wasNull(), and proceed accordingly. It's not pretty, but it works. Here's an example:

```

int age = rs.getInt("age");
if (!rs.wasNull())
out.println ("Age: " + age);

```

Another way to check for null values is to use the `getObject()` method. If a column is null, `getObject()` always returns null. Using `getObject()` can eliminate the need to call `wasNull()` and result in simpler code, but the objects returned aren't easy-to-use primitives.

Updating the Database

Most database-enabled web sites need to do more than just perform queries. When a client submits an order or provides some kind of information, the data needs to be entered into the database. When you know you're executing a SQL UPDATE, INSERT, or DELETE statement and you know you don't expect a `ResultSet`, you can use the `executeUpdate()` method of `Statement`. It returns a count that indicates the number of rows modified by the statement. It's used like this:

```
int count = stmt.executeUpdate("DELETE FROM CUSTOMERS WHERE CUSTOMER_ID = 5");
```

If you are executing SQL that may return either a `ResultSet` or a count (say, if you're handling user-submitted SQL or building generic data-handling classes), use the generic `execute()` method of `Statement`. It returns a boolean whose value is true if the SQL statement produced one or more `ResultSet` objects or false if it resulted in an update count:

```
boolean b = stmt.execute(sql);
```

The `getResultSet()` and `getUpdateCount()` methods of `Statement` provide access to the results of the `execute()` method. Example -3 demonstrates the use of these methods which prints the records from any kind of SQL statement.

Example-3. ExecuteSQL.java

```

import java.sql.*;
import java.io.*;
/** A general-purpose SQL interpreter program. */
public class ExecuteSQL {
public static void main(String[ ] args)
{
Connection conn = null; //Our JDBC connection to the database server
try { String driver = null, url = null, user = "", password = "";
for(int n = 0; n < args.length; n++) {
if(args[n].equals("-d")) driver = args[++n];
else if(args[n].equals("-u")) user = args[++n];
else if(args[n].equals("-p")) password = args[++n];
else if(url == null) url = args[n];
else throw new IllegalArgumentException("Unknown argument.");
}
// The only required argument is the database URL.

```

```

if(url == null)
throw new IllegalArgumentException("No database specified");
// If the user specified the classname for the DB driver, load
// that class dynamically. This gives the driver the opportunity
// to register itself with the DriverManager.

if(driver != null) Class.forName(driver);
// Now open a connection to the specified database, using the user-specified username and
// password, if any. The driver manager will try all of the DB drivers //it knows about to try to parse
//the URL and connect to the DB server.
conn = DriverManager.getConnection(url, user, password);
// Now create the statement object we'll use to talk to the DB
Statement s = conn.createStatement();
// Get a stream to read from the console
BufferedReader in =new BufferedReader(new InputStreamReader(System.in));
// Loop forever, reading the user's queries and executing them
while(true) {
System.out.print("sql> "); // prompt the user System.out.flush(); //make the prompt appear now.
String sql = in.readLine(); //get a line of input from user // Quit when the user types "quit".
if((sql == null) // sql.equals ("quit")) break; // Ignore blank lines
if(sql.length( ) == 0) continue;
// Now, execute the user's line of SQL and display results.
try {
// We don't know if this is a query or some kind of// update, so we use execute( ) instead of
executeQuery( )// or executeUpdate( ). If the return value is true, it was // a query, else an update.
boolean status = s.execute(sql);
// Some complex SQL queries can return more than one set of results, so loop until there are no
more results
do {
if(status) { // it was a query and returns a ResultSet
ResultSet rs = s.getResultSet();
// Get results printResultsTable(rs, System.out); // Display them }
else {
// If the SQL command that was executed was some kind of update rather than a query, then it
// doesn't return a ResultSet. Instead, we just print the number of rows that were affected. int
numUpdates = s.getUpdateCount();
System.out.println ("Ok. " + numUpdates + " rows affected.");
}
// Now go see if there are even more results, and // continue the results display loop if there are.
status = s.getMoreResults( );
} while (status // s.getUpdateCount ( ) != -1); }

```

```

// If a SQLException is thrown, display an error message. Note that SQLExceptions can have a
// general message and a DB-specific message returned by getSQLState( )

catch (SQLException e) {
    System.err.println ("SQLException: " + e.getMessage () + ":" + e.getSQLState ());
}

// Each time through this loop, check to see if there were any // warnings. Note that there can be a
// whole chain of warnings.
finally {
    // print out any warnings that occurred SQLWarning w;
    for(w=conn.getWarnings( ); w != null; w=w.getNextWarning( ))
        System.err.println("WARNING: " + w.getMessage() + ":" + w.getSQLState());
}

// Handle exceptions that occur during argument parsing, database
// connection setup, etc. For SQLExceptions, print the details.
catch (Exception e) {
    System.err.println(e);
    if(e instanceof SQLException)
        System.err.println ("SQL State: " + ((SQLException)e).getSQLState());
    System.err.println ("Usage: java ExecuteSQL [-d <driver>] " + "[-u <user>] [-p <password>]
<database URL>"); }

// Be sure to always close the database connection when we exit, // whether we exit because the user
// types 'quit' or because of an // exception thrown while setting things up. Closing this connection //
// also implicitly closes any open statements and result sets // associated with it.
finally {
    try { conn.close( ); } catch (Exception e) { }
}

/** This method attempts to output the contents of a ResultSet in a * textual table. It relies on the
ResultSetMetaData class, but a fair * bit of the code is simple string manipulation. **/



static void printResultsTable (ResultSet rs, OutputStream output) throws SQLException
{
    // Set up the output stream
    PrintWriter out = new PrintWriter(output);

    // Get some "meta data" (column names, etc.) about the results
    ResultSetMetaData metadata = rs.getMetaData( );

    // Variables to hold important data about the table to be displayed
    int numcols = metadata.getColumnCount ( ); // how many columns
    String [ ] labels = new String [numcols]; // the column labels
}

```

```

int[ ] colwidths = new int[numcols]; // the width of each
int[ ] colpos = new int[numcols];
// start position of each
int linewidth;
// total width of table Figure out how wide the columns are, where each one begins,
// how wide each row of the table will be, etc.
linewidth = 1; // for the initial '|'.
for(int i = 0; i < numcols; i++) {
    // for each column colpos[i] = linewidth; // save its position
    labels[i] = metadata.getColumnLabel(i+1);
    // get its label // Get the column width. If the db doesn't report one, guess
    // 30 characters. Then check the length of the label, and use // it if it is larger than the column width
    int size = metadata getColumnDisplaySize(i+1);
    if(size == -1) size = 30;
    // Some drivers return -1...
    if(size > 500) size = 30; // Don't allow unreasonable sizes
    int labelsize = labels[i].length();
    if(labelsize > size) size = labelsize;
    colwidths[i] = size + 1; // save the column size
    linewidth += colwidths[i] + 2; // increment total size }
    // Create a horizontal divider line we use in the table.
    // Also create a blank line that is the initial value of each // line of the table
    StringBuffer divider = new StringBuffer(linewidth);
    StringBuffer blankline = new StringBuffer(linewidth);
    for (int i = 0; i < linewidth; i++) {
        divider.insert(i, '-');
        blankline.insert(i, " ");
    }
    // Put special marks in the divider line at the column positions
    for (int i=0; i<numcols; i++)
        divider.setCharAt(colpos[i]-1, '+');
        divider.setCharAt(linewidth-1, '+');
    // Begin the table output with a divider line
    out.println(divider);
    // The next line of the table contains the column labels. Begin with a blank line, and put the column
    // names and column divider characters "/" into it. overwrite( ) is defined below.
    StringBuffer line = new StringBuffer(blankline.toString()); line.setCharAt(0, '/');
    for(int i = 0; i < numcols; i++) {
        int pos = colpos[i] + 1 + (colwidths[i]-labels[i].length())/2;
        overwrite(line, pos, labels[i]);
        overwrite(line, colpos[i] + colwidths[i], "/");
    }
}

```

```

// Then output the line of column labels and another divider
out.println(line);
out.println(divider);
// Now, output the table data. Loop through the ResultSet, using // the next( ) method to get the rows
one at a time. Obtain the // value of each column with getObject( ), and output it, much as // we did
for the column labels above.

while(rs.next( )) {
    line = new StringBuffer(blankline.toString( ));
    line.setCharAt(0, '/');
    for(int i = 0; i < numcols; i++) {
        Object value = rs.getObject(i+1);
        if(value != null)
            overwrite(line, colpos[i] + 1, value.toString( ).trim( ));
        overwrite(line, colpos[i] + colwidths[i], " /");
    }
    out.println(line);
}
// Finally, end the table with one last divider line.
out.println(divider);
out.flush( );
}

/** This utility method is used when printing the table of results */
static void overwrite(StringBuffer b, int pos, String s) {
    int slen = s.length( ); // string length
    int blen = b.length( ); // buffer length
    if(pos+slen > blen) slen = blen-pos; // does it fit?
    for(int i = 0; i < slen; i++) // copy string into buffer
        b.setCharAt (pos+i, s.charAt (i));
}
}

```

This example uses execute () to execute whatever SQL statement is passed to the HtmlSQLResult constructor. Then, depending on the return value, it either calls getResultSet() or getUpdateCount(). Note that neither getResultSet () nor getUpdateCount () should be called more than once per query.

Using Prepared Statements

A PreparedStatement object is like a regular Statement object, in that it can be used to execute SQL statements. The important difference is that the SQL in a PreparedStatement is precompiled by the database for faster execution. Once a PreparedStatement has been compiled, it can still be

customized by adjusting predefined parameters. Prepared statements are useful in applications that have to run the same general SQL command over and over.

Use the `prepareStatement(String)` method of `Connection` to create `PreparedStatement` objects. Use the `?` character as a placeholder for values to be substituted later.

For example:

```
PreparedStatement pstmt = con.prepareStatement( "INSERT INTO ORDERS (ORDER_ID,  
CUSTOMER_ID, TOTAL) VALUES (?, ?, ?)" ); // Other code  
pstmt.clearParameters(); // clear any previous parameter values  
pstmt.setInt(1, 2); // set ORDER_ID  
pstmt.setInt(2, 4); // set CUSTOMER_ID  
pstmt.setDouble(3, 53.43); // set TOTAL  
pstmt.executeUpdate(); // execute the stored SQL
```

The `clearParameters()` method removes any previously defined parameter values, while the `setXXX()` methods are used to assign actual values to each of the placeholder question marks. Once you have assigned values for all the parameters, call `executeUpdate()` to execute the `PreparedStatement`. The `PreparedStatement` class has an important application in conjunction with servlets. When loading user-submitted text into the database using `Statement` objects and dynamic SQL, you must be careful not to accidentally introduce any SQL control characters (such as `"` or `'`) without escaping them in the manner required by your database. With a database like Oracle that surrounds strings with single quotes, an attempt to insert John d'Artagan into the database results in this corrupted SQL:

```
INSERT INTO MUSKETEERS (NAME) VALUES ('John d'Artagan')
```

As you can see, the string terminates twice. One solution is to manually replace the single quote `'` with two single quotes `("")`, the Oracle escape sequence for one single quote. This solution requires you to escape every character that your database treats as special—not easy and not consistent with writing platform-independent code. A far better solution is to use a `PreparedStatement` and pass the string using its `setString()` method, as shown next. The `PreparedStatement` automatically escapes the string as necessary for your database:

```
PreparedStatement pstmt = con.prepareStatement( "INSERT INTO MUSKETEERS (NAME)  
VALUES (?)"); pstmt.setString(1, "John d'Artagan");  
pstmt.executeUpdate();
```

Using Database Metadata

Sometimes, in addition to querying and updating the data in a database, you also want to retrieve information about the database itself and its contents. This information is called metadata. The `DatabaseMetaData` interface allows you to retrieve this kind of information. You can obtain an

object that implements this interface by calling the `getMetaData()` method of the `Connection` object, as shown in Example -4.

After `GetDBInfo` opens a database Connection and obtains a `DatabaseMetaData` object, it displays some general information about the database server and JDBC driver. Then, if the user just specified a database name on the command line, the program lists all the tables in that database. If the user specified a database name and a table name, however, the program lists the name and data type of each column in that table.

An interesting feature of this `GetDBInfo` program is how it obtains the parameters needed to connect to the database. The example operates on the premise that at any given site, it is typically used to connect to the same database server, using the same database driver, and may also be used with the same database username and password. So, instead of requiring the user to type all this cumbersome information on the command line each time the program is run, the program reads default values from a file named `DB.props` that is stored in the same directory as the `GetDBInfo.class` file. In order to run Example 4, you have to create an appropriate `DB.props` file for your system. On my system, this file contains:

```
# The name of the JDBC driver class  
driver=com.mysql.jdbc.Driver  
# The URL that specifies the database server.  
# It should not include the name of the database to connect to server=jdbc:mysql:///  
# The database account name  
user=david  
# The password for the specified account, if any. # Uncomment the line below if you need to  
specify a password #password=
```

Lines that begin with `#` are comments, obviously. The `name=value` format is the standard file format for the `java.util.Properties` object that is used to read the contents of this file. After the program reads the default values from the `DB.props` file, it parses its command-line arguments, which can override the driver, server, user, and password properties specified in the file. The name of the database to connect to must be specified on the command line; the database name is simply appended to the server URL. The name of a table in the database can optionally be specified on the command line. For example, you might run the program as follows:

```
% java GetDBInfo api class  
DBMS: MySQL 4.0.14-standard  
JDBC Driver: MySQL-AB JDBC Driver 3.0.8-stable ($Date: 2004/01/29 23:10:56 $) Database:  
jdbc:mysql:///apidb  
User: david@localhost  
Columns of class:  
id : int  
packageId : int  
name : varchar
```

Example 4. GetDBInfo.java

```
import java.sql.*;
import java.util.Properties;
/** This class uses the DatabaseMetaData class to obtain information about the database, the
JDBC driver, and the tables in the database, or about the columns of a named table. **/
public class GetDBInfo {
public static void main(String[ ] args) {
Connection c = null; // The JDBC connection to the database server
try {
// Look for the properties file DB.props in the same directory as this program. It will contain
// default values for the various parameters needed to connect to a database
Properties p = new Properties( );
try {
p.load(GetDBInfo.class.getResourceAsStream("DB.props"));
} catch (Exception e) { }
// Get default values from the properties file
String driver = p.getProperty("driver"); // Driver class name
String server = p.getProperty("server", ""); // JDBC URL for server
String user = p.getProperty("user", ""); // db user name
String password = p.getProperty("password", ""); // db password
// These variables don't have defaults

String database = null; // The db name (appended to server URL)
String table = null; // The optional name of a table in the db
// Parse the command-line args to override the default values above
for(int i = 0; i < args.length; i++) {
if(args[i].equals("-d")) driver = args[++i]; // -d <driver>
else if(args[i].equals("-s")) server = args[++i]; // -s <server>
else if(args[i].equals("-u")) user = args[++i]; // -u <user>
else if(args[i].equals("-p")) password = args[++i];
else if(database == null) database = args[i]; // <dbname>
else if(table == null) table = args[i]; // <table>
else throw new IllegalArgumentException("Unknown argument: " +args[i]);
}
// Make sure that at least a server or a database were specified. // If not, we have no idea what to
connect to, and cannot continue.
if((server.length( ) == 0) && (database.length( ) == 0))
throw new IllegalArgumentException("No database specified.");
// Load the db driver, if any was specified.
if(driver != null)
Class.forName(driver);
```

```

// Now attempt to open a connection to the specified database on the specified server, using the
//specified name and password
c = DriverManager.getConnection(server+database, user, password);
// Get the DatabaseMetaData object for the connection.
//This is the object that will return us all the data we're interested in here
DatabaseMetaData md = c.getMetaData( ); // Display information about the server, the driver, etc.
System.out.println("DBMS: " + md.getDatabaseProductName( ) + " " +
md.getDatabaseProductVersion( ));
System.out.println("JDBC Driver: " + md.getDriverName( ) + " " + md.getDriverVersion( ));
System.out.println("Database: " + md.getURL( ));
System.out.println("User: " + md.getUserName( ));
// Now, if the user did not specify a table, then display a list of all tables defined in the named
//database. Note that tables are returned in a ResultSet, just like query results are.
if(table == null) {
System.out.println("Tables:");
ResultSet r = md.getTables("", "", "%", null);
while(r.next( ))
System.out.println("\t" + r.getString(3));
}
// Otherwise, list all columns of the specified table.
// Again, information about the columns is returned in a ResultSet
else {
System.out.println("Columns of " + table + ": ");
ResultSet r = md.getColumns("", "", table, "%");
while(r.next( ))
System.out.println("\t" + r.getString(4) + ":" + r.getString(6));
}
}

// Print an error message if anything goes wrong. catch (Exception e) {
System.err.println(e);
if(e instanceof SQLException)
System.err.println(((SQLException)e).getSQLState( ));
System.err.println("Usage: java GetDBInfo [-d <driver>] " + "[-s <dbserver>]\n" +
"\t[-u <username>] [-p <password>] <dbname>"); }
// Always remember to close the Connection object when we're done!
finally {
try { c.close( ); } catch (Exception e) { }
}
}
}

```

Building a Database

Example -5 shows a program, MakeAPIDB, that takes a list of class names and uses the Java Reflection API to build a database of those classes, the packages they belong to, and all methods and fields defined by the classes. Example 4 shows a program that uses the database created by this example.

MakeAPIDB uses the SQL CREATE TABLE statement to add three tables, named package, class, and member, to the database. The program then inserts data into those tables using INSERT INTO statements. The program uses the same INSERT INTO statements repeatedly, as it iterates through the list of class names. In this type of situation, you can often increase the efficiency of your insertions if you use PreparedStatement objects to execute the statements.

A prepared statement is essentially a blueprint for the statements you need to execute. When you send a SQL statement to the database, the database interprets the SQL and creates a template for executing the statement. If you are sending the same SQL statement repeatedly, only with different input parameters, the database still has to interpret the SQL each time. On database platforms that support prepared statements, you can eliminate this inefficiency by sending a prepared statement to the database before you actually make any calls to the database. The database interprets the prepared statement and creates its template just once. Then, when you execute the prepared statement repeatedly with different input parameters, the database uses the template it has already created.

JDBC provides the PreparedStatement class to support prepared statements, but it doesn't guarantee that the underlying database actually takes advantage of them. You create a PreparedStatement with the prepareStatement() method of a Connection object, as shown in Example 5.

MakeAPIDB passes a SQL statement to prepareStatement(), substituting ? placeholders for the variable parameters in the statement. Later, before the program executes the prepared statement, it binds values to these parameters using the various setX() methods (e.g., setInt() and setString()) of the PreparedStatement object. Each setX() method takes two arguments: a parameter index (starting with 1) and a value. Then the program calls the executeUpdate() method of the PreparedStatement to execute the statement.

(PreparedStatement also provides execute() and executeQuery() methods, just like Statement.)

MakeAPIDB expects its first argument to be the name of a file that contains a list of classes to be placed into the database. The classes should be listed one to a line; each line must contain a fully qualified class name (i.e., it must specify both package name and class name). Such a file might contain lines like the following:

```
java.applet.Applet  
java.applet.AppletContext  
java.applet.AppletStub ...  
java.util.zip.ZipOutputStream
```

The program reads database parameters from a Properties file named APIDB.props in the current directory or from an alternate Properties file specified as the second command-line argument. This Properties file is similar to, but not quite the same as, the one used in conjunction with Example - 4; it should contain properties named driver, database, user, and password. On my system, the APIDB.props file looks as follows:

```
# The full classname of the JDBC driver to load: this is a MySQL driver  
driver=com.mysql.jdbc.Driver  
# The URL of the mysql server (localhost) and database (apidb) to connect to  
database=jdbc:mysql:///apidb  
# The name of the database user account  
user=david  
# The password for the database user account.  
# Uncomment the line below to specify a password  
#password=
```

Note that before you run this program, you must create the database for it on your database server. To do this, you have to follow the instructions provided by your database vendor. You can also use an existing database, as long as it doesn't already contain tables named package, class, or member.

Example -5. MakeAPIDB.java

```
package je3.sql;  
import java.sql.*;  
import java.lang.reflect.*;  
import java.io.*;  
import java.util.*;  
/**  
 * This class is a standalone program that reads a list of classes and  
 * builds a database of packages, classes, and class fields and methods.  
 */  
public class MakeAPIDB {  
    public static void main(String args[ ]) {  
        Connection c = null; // The connection to the database  
        try { // Read the classes to index from a file specified by args[0]  
            ArrayList classnames = new ArrayList( );  
            BufferedReader in = new BufferedReader(new FileReader(args[0]));  
            String name;  
            while((name = in.readLine( )) != null) classnames.add(name);  
            // Now determine the values needed to set up the database // connection. The program attempts to  
            //read a property file named "APIDB.props", or one optionally specified by args[1]. This property  
            //file (if any) may contain "driver", "database", "user", and "password" properties that specify the  
            //necessary values for connecting to the db. If the properties file does not exist, or  
            //does not contain the named properties, defaults will be used.  
        } catch (Exception e) {  
            System.out.println("Error: " + e);  
        }  
    }  
}
```

```

Properties p = new Properties( ); // Empty properties
try {
p.load(new FileInputStream(args[1])); // Try to load properties
}
catch (Exception e1) {
try { p.load(new FileInputStream("APIDB.props")); }
catch (Exception e2) { }
}
// Read values from Properties file
String driver = p.getProperty("driver");
String database = p.getProperty("database");
String user = p.getProperty("user", "");
String password = p.getProperty("password", "");
// The driver and database properties are mandatory
if (driver == null)
throw new IllegalArgumentException("No driver specified!");
if (database == null)
throw new IllegalArgumentException("No database specified!");
// Load the driver. It registers itself with DriverManager.
Class.forName(driver);
// And set up a connection to the specified database
c = DriverManager.getConnection(database, user, password);
// Create three new tables for our data
// The package table contains a package id and a package name.
// The class table contains a class id, a package id, and a name.
// The member table contains a class id, a member name, and a bit
// that indicates whether the class member is a field or a method.
Statement s = c.createStatement( );
s.executeUpdate("CREATE TABLE package " + "(id INT, name VARCHAR(80))");
s.executeUpdate("CREATE TABLE class " + "(id INT, packageId INT, name VARCHAR(48))");
s.executeUpdate("CREATE TABLE member " + "(classId INT, name VARCHAR(48), isField BIT)");
// Prepare some statements that will be used to insert records into // these three tables.
insertpackage = c.prepareStatement("INSERT INTO package VALUES(?,?)");
insertclass = c.prepareStatement("INSERT INTO class VALUES(?,?,?)");
insertmember = c.prepareStatement("INSERT INTO member VALUES(?,?,?,?)");
// Now loop through the list of classes and use reflection // to store them all in the tables
int numclasses = classnames.size( );
for(int i = 0; i < numclasses; i++) {
try {
storeClass((String)classnames.get(i));
}

```

```

catch(ClassNotFoundException e) {
    System.out.println("WARNING: class not found: " + classnames.get(i) + "; SKIPPING");
}
}

catch (Exception e) {
    System.err.println(e);
    if(e instanceof SQLException)
        System.err.println("SQLState: " + ((SQLException)e).getSQLState( ));
    System.err.println("Usage: java MakeAPIDB " + "<classlistfile> <propfile>");
}

// When we're done, close the connection to the database
finally {
    try { c.close( ); }
    catch (Exception e) { } }
/***
 * This hash table records the mapping between package names and package
 * id. This is the only one we need to store temporarily. The others are
 * stored in the db and don't have to be looked up by this program ***/
static Map package_to_id = new HashMap( );
// Counters for the package and class identifier columns
static int packageId = 0, classId = 0;
// Some prepared SQL statements for use in inserting
// new values into the tables. Initialized in main( ) above.
static PreparedStatement insertpackage, insertclass, insertmember;
/***
 * Given a fully-qualified classname, this method stores the package name in the package table (if it
 * is not already there), stores the class name in the class table, and then uses the Java Reflection
 * API to look up all methods and fields of the class, and stores those in the member table. ***/
public static void storeClass(String name) throws SQLException, ClassNotFoundException {
    String packagename, classname;
    // Dynamically load the class.
    Class c = Class.forName(name);

    // Display output so the user knows that the program is progressing
    System.out.println("Storing data for: " + name); // Figure out the packagename and the classname
    int pos = name.lastIndexOf('.');
    if(pos == -1) {
        packagename = "";
        classname = name;
    }
    else {

```

```

packagename = name.substring(0,pos);
classname = name.substring(pos+1);
}
// Figure out what the package id is. If there is one, then this package has already been stored in
// the database. Otherwise, assign an id, and store it and the packagename in the db.
Integer pid;
pid = (Integer)package_to_id.get(packagename); // Check hashtable
if(pid == null) {
    pid = new Integer(++packageId); // Assign an id
    package_to_id.put(packagename, pid); // Remember it
    insertpackage.setInt(1, packageId); // Set statement args
    insertpackage.setString(2, packagename);
    insertpackage.executeUpdate(); // Insert into package db
}
// Now, store the classname in the class table of the database. This record includes the package id,
// so that the class is linked to/ the package that contains it. To store the class, we set arguments
// to the PreparedStatement, then execute that statement
insertclass.setInt(1, ++classId); // Set class identifier
insertclass.setInt(2, pid.intValue()); // Set package identifier
insertclass.setString(3, classname); // Set class name
insertclass.executeUpdate(); // Insert the class record
// Now, get a list of all non-private methods of the class, and insert those into the "members" table
// of the database. Each record includes the class id of the containing class, and also a value that
// indicates that these are methods, not fields.

```

```

Method[ ] methods = c.getDeclaredMethods(); // Get a list of methods
for(int i = 0; i < methods.length; i++) {
    // For all non-private
    if(Modifier.isPrivate(methods[i].getModifiers()))
        continue;
    insertmember.setInt(1, classId); // Set the class id
    insertmember.setString(2, methods[i].getName()); // Set method name
    insertmember.setBoolean(3, false); // It is not a field
    insertmember.executeUpdate(); // Insert into db
}
// Do the same thing for the non-private fields of the class
Field[ ] fields = c.getDeclaredFields(); // Get a list of fields
for(int i = 0; i < fields.length; i++) { // For each non-private
    if(Modifier.isPrivate(fields[i].getModifiers())) continue;
    insertmember.setInt(1, classId); // Set the class id
    insertmember.setString(2, fields[i].getName()); // Set field name
    insertmember.setBoolean(3, true); // It is a field
}

```

```
insertmember.executeUpdate( ); // Insert the record  
}  
}  
}
```

Using the API Database

Example -6 displays a program, LookupAPI, that uses the database built by the MakeAPIDB program of Example -5 and makes interesting SQL queries against it. LookupAPI behaves as follows:

- When invoked with the name of a class member, it lists the full name (including package and class) of each field and/or method that has that name.
- When run with the name of a class, it lists the full name of every class in any package that has that name.
- When called with a portion of a package name, it lists the names of all the packages that contain that string.
- When invoked with the -l option and a class name, it lists every member of every class that has that name.
- When run with the -l option and a portion of a package name, it lists all the classes and interfaces in any package that matches that string.

LookupAPI reads the same APIDB.props property file MakeAPIDB does. Or, alternatively, it reads a property file specified on the command line following a -p flag. Using the database connection parameters in the property file, the program connects to a database and executes the necessary SQL queries to return the desired information. Note that it calls the setReadOnly() method of the Connection object. Doing this provides a hint that the program performs only queries and doesn't modify the database in any way. For some database systems, this may improve efficiency.

Other than the setReadOnly() method, this example doesn't introduce any new JDBC features. It simply serves as a real-world application of a database and demonstrates some of the powerful database queries that can be expressed using SQL.

Example -6. LookupAPI.java

```
import java.sql.*;  
import java.io.FileInputStream;  
import java.util.Properties;
```

```
/**
```

```
* This program uses the database created by MakeAPIDB. It opens a connection to a database  
* using the same property file used by MakeAPIDB. Then it queries that database in several
```

*interesting ways to obtain useful information about Java APIs. It can be used to look up the fully-qualified name of a member, class, or package, or it can be used to list the members of a class or package. **/

```

public class LookupAPI {
    public static void main(String[ ] args) {
        Connection c = null; // JDBC connection to the database
        try {
            // Some default values
            String target = null; // The name to look up
            boolean list = false; // List members or lookup name?
            String propfile = "APIDB.props"; // The file of db parameters,
            //Parse the command-line arguments
            for(int i = 0; i < args.length; i++) {
                if(args[i].equals("-l")) list = true;
                else if(args[i].equals("-p")) propfile = args[++i];
                else if(target != null)
                    throw new IllegalArgumentException("Unexpected argument: " + args[i]);
                else target = args[i];
            }
            if(target == null)
                throw new IllegalArgumentException("No target specified");
            // Now determine the values needed to set up the database connection. The program attempts to
            //read a property file named "APIDB.props", or optionally specified with the -p argument. This
            //property file may contain "driver", "database", "user", and "password" properties that
            //specify the necessary values for connecting to the db. If the properties file does not exist, or does
            //not contain the named properties, defaults will be used.
        }
    }
}

```

```

Properties p = new Properties( ); // Empty properties
try { p.load(new FileInputStream(propfile)); } // Try to load props
catch (Exception e) { }
// Read values from Properties file
String driver = p.getProperty("driver");
String database = p.getProperty("database");
String user = p.getProperty("user", "");
String password = p.getProperty("password", ""); // The driver and database properties are
mandatory if(driver == null)
throw new IllegalArgumentException("No driver specified!");
if(database == null)
throw new IllegalArgumentException("No database specified!"); // Load the database driver
Class.forName(driver);
// And set up a connection to the specified database
c = DriverManager.getConnection(database, user, password);

```

```

// Tell it we will not do any updates. // This hint may improve efficiency.
c.setReadOnly(true);

// If the "-l" option was given, then list the members of the named package or class. Otherwise,
// lookup all matches for the specified member, class, or package.
if (list) list(c, target);
else lookup(c, target);
}

// If anything goes wrong, print the exception and a usage message. If // a SQLException is thrown,
// display the state message it includes.
catch (Exception e) {
System.out.println(e);
if (e instanceof SQLException)
System.out.println(((SQLException) e).getSQLState());
System.out.println("Usage: java LookupAPI [-l] [-p <propfile>] " + "target");
}

// Always close the DB connection when we're done with it.
finally {
try { c.close(); } catch (Exception e) { }
}
}

/***
* This method looks up all matches for the specified target string in the database. First, it prints the
* full name of any members by that name. Then it prints the full name of any classes by that name.
* Then it prints the name of any packages that contain the specified name ***/
public static void lookup(Connection c, String target) throws SQLException {
// Create the Statement object we'll use to query the database
Statement s = c.createStatement();
// Go find all class members with the specified name
s.executeQuery("SELECT DISTINCT " + "package.name, class.name, member.name,
member.isField" + " FROM package, class, member" + " WHERE member.name=" + target + """
+ " AND member.classId=class.id " + " AND class.packageId=package.id");
// Loop through the results, and print them out (if there are any).
ResultSet r = s.getResultSet();
while(r.next( )) {
String pkg = r.getString(1); // package name
String cls = r.getString(2); // class name
String member = r.getString(3); // member name
boolean isField = r.getBoolean(4); // is the member a field?

// Display this match
System.out.println(pkg + "." + cls + "." + member + (isField?"":"(" )"));
}
}

```

```

// Now look for a class with the specified name
s.executeQuery("SELECT package.name, class.name " + "FROM package, class " +
"WHERE class.name='" + target + "' " + "AND class.packageId=package.id");
// Loop through the results and print them out
r = s.getResultSet( );
while(r.next( )) System.out.println(r.getString(1) + "." + r.getString(2));
// Finally, look for a package that matches a part of the name.
// Note the use of the SQL LIKE keyword and % wildcard characters
s.executeQuery("SELECT name FROM package " + "WHERE name='" + target + "' " +
"OR name LIKE '%." + target + "%' " + "OR name LIKE '" + target + "%' " +
"OR name LIKE '%." + target + "%'");
// Loop through the results and print them out
r = s.getResultSet( );
while(r.next( ))
System.out.println(r.getString(1)); // Finally, close the Statement object
s.close( );
}

/**
 * This method looks for classes with the specified name, or packages that contain the specified
 * name. For each class it finds, it displays all methods and fields of the class. For each package it
 * finds, it displays all classes in the package. */

```

```

public static void list(Connection conn, String target) throws SQLException {
// Create two Statement objects to query the database with
Statement s = conn.createStatement( );
Statement t = conn.createStatement( );
// Look for a class with the given name
s.executeQuery("SELECT package.name, class.name " + "FROM package, class " +
"WHERE class.name='" + target + "' " + "AND class.packageId=package.id");
// Loop through all matches
ResultSet r = s.getResultSet( );
while(r.next( )) {
String p = r.getString(1); // package name
String c = r.getString(2); // class name
// Print out the matching class name
System.out.println("class " + p + "." + c + " {"); // Now query all members of the class
t.executeQuery("SELECT DISTINCT member.name, member.isField " + "FROM package, class,
member " + "WHERE package.name = '" + p + "' " + "AND class.name = '" + c + "' " +
"AND member.classId=class.id " + "AND class.packageId=package.id " +
"ORDER BY member.isField, member.name");
// Loop through the ordered list of all members, and print them out
ResultSet r2 = t.getResultSet( );

```

```

while(r2.next( )) {
    String m = r2.getString(1);
    int isField = r2.getInt(2);
    System.out.println(" " + m + ((isField == 1)?"";"( )")); }
    // End the class listing
    System.out.println("}");
}

// Now go look for a package that matches the specified name
s.executeQuery("SELECT name FROM package " + "WHERE name=" + target + "'''' +
    " OR name LIKE '%." + target + "%' " + " OR name LIKE '" + target + "%' " +
    " OR name LIKE '%." + target + "'''");
// Loop through any matching packages
r = s.getResultSet( );
while(r.next( )) {
    // Display the name of the package
    String p = r.getString(1);
    System.out.println("Package " + p + ": ");
    // Get a list of all classes and interfaces in the package
    t.executeQuery("SELECT class.name FROM package, class " + "WHERE package.name=" + p +
        "''' " + "AND class.packageId=package.id " + "ORDER BY class.name");
    // Loop through the list and print them out.
    ResultSet r2 = t.getResultSet( );
    while(r2.next( ))
        System.out.println(" " + r2.getString(1));
    // Finally, close both Statement objects
    s.close( ); t.close( );
}
}

```

Atomic Transactions

By default, a newly created database Connection object is in auto-commit mode. That means that each update to the database is treated as a separate transaction and is automatically committed to the database. Sometimes, however, you want to group several updates into a single atomic transaction, with the property that either all the updates complete successfully or no updates occur at all. With a database system (and JDBC driver) that supports it, you can take the Connection out of autocommit mode and explicitly call commit() to commit a batch of transactions or call rollback() to abort a batch of transactions, undoing the ones that have already been done.

Performing Batch Updates

One way that you can improve the performance of your JDBC application is to execute a number of SQL commands in a batch. With batch execution, you add any number of SQL commands to the statement. The statement holds these SQL commands in memory until you tell it that you are ready for the database to execute the SQL. When you call `executeBatch()`, the statement sends the entire batch of SQL in one network communication. In addition to the `executeBatch()` method, two other methods are needed for batch execution:

- `public void addBatch(String):` Adds a SQL command to the current batch of commands for the Statement object.
- `public void clearBatch():` Makes the set of commands in the current batch empty.

The use of batch updating is straightforward. You add SQL commands to the statement with the `addBatch(String)` command. When you are ready for the commands to be executed, you call the `executeBatch()` method. This causes the statement to send all the SQL commands to the database for execution. In code, it looks like this:

```
// The variable someSqlCommand in the method calls is a SQL command
stmt.addBatch(someSqlCommand);
stmt.addBatch(someSqlCommand);
stmt.addBatch(someSqlCommand);
stmt.addBatch(someSqlCommand);
int[] results = stmt.executeBatch();
```

As you can see in this snippet, the `executeBatch()` method returns an int array that contains the number of rows affected by each of the commands. The result of the first SQL command that was added to the statement is returned in the first element of the array, the result of the second SQL command is in the second element, and so on. Since the `executeBatch()` method returns an int array, the one type of SQL command that cannot be executed by batching is a SQL SELECT command, which returns a `ResultSet` object, not an int.

Self-Practice Program – 1

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
class Employee
{
public static void main (String args []) throws SQLException
{
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
System.out.println("Driver loaded successfully"); // Connect to the database
// You must put a database name after the @ sign in the connection URL.
// You can use either the fully specified SQL*net syntax or a short cut syntax as
//<host>:<port>:<sid>. The example uses the short cut syntax. String url =
//jdbc:oracle:thin:@dlsun511:1721:dbms733"; //String url = "jdbc:oracle:thin:@oracle";
String userName = "scott";
String password = "tiger";
if (args.length > 0) url = args[0];
if (args.length > 1) userName = args[1];
if (args.length > 2) password = args[2];

System.out.println("Trying to connecting database..... . . .");
Connection conn = DriverManager.getConnection (url, userName, password);
System.out.println("Connection established."); // Create a Statement
Statement stmt = conn.createStatement ();
// Select the ENAME column from the EMP table
ResultSet rset = stmt.executeQuery ("select ENAME from EMP");
// Iterate through the result and print the employee names
while (rset.next ())
System.out.println (rset.getString (1));
}
}
```

Self-Practice Program – 2

```
/**
* This sample can be used to check the JDBC installation. Just run it and provide the connect
*information. It will select * "Hello World" from the database. */
// You need to import the java.sql package to use JDBC
import java.sql.*;
import java.io.*;
```

```

class JdbcCheckup
{
    public static void main (String args [])
        throws SQLException, IOException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
        // Prompt the user for connect information
        System.out.println ("Please enter information to test connection to the database");
        String user;
        String password;
        String database;
        user = readEntry ("user: ");
        int slash_index = user.indexOf('/');
        if(slash_index != -1)
        {
            password = user.substring (slash_index + 1);
            user = user.substring (0, slash_index);
        }
        else
            password = readEntry ("password: ");
        database = readEntry ("database (a TNSNAME entry): ");
        System.out.print ("Connecting to the database...");
        System.out.flush ();

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        System.out.println ("Connecting... ");
        String url = "jdbc:oracle:oci8:@ " + database;
        Connection conn = DriverManager.getConnection (url, user, password);
        System.out.println ("connected.");
        // Create a statement
        Statement stmt = conn.createStatement ();
        // Do the SQL "Hello World" thing
        ResultSet rset = stmt.executeQuery ("select 'Hello World' from dual");
        while (rset.next ())
            System.out.println (rset.getString (1));
        System.out.println ("Your JDBC installation is correct.");
        rset.close(); // close the resultSet
        stmt.close(); // Close the statement
        conn.close(); // Close the connection
    }
}

```

```

// Utility function to read a line from standard input static String readEntry (String prompt)
{
try
{
StringBuffer buffer = new StringBuffer ();
System.out.print (prompt);
System.out.flush ();
int c = System.in.read ();
while (c != '\n' && c != -1)
{
buffer.append ((char)c);
c = System.in.read ();
}
return buffer.toString ().trim ();
}
catch (IOException e)
{
return "";
}
}
}
}

```

Self-Practice Program – 3

```

/***
* This sample shows how to call a PL/SQL stored procedure using the SQL92 syntax. See also the
* other sample PLSQL.java. */

```

```

import java.sql.*;
import java.io.*;
class PLSQLExample
{
public static void main (String args [])
throws SQLException, IOException
{
// Load the driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
String url = "jdbc:oracle:oci8:@";
try {
String url1 = System.getProperty("JDBC_URL");
if(url1 != null)
url = url1;

```

```

} catch (Exception e) {
// If there is any security exception, ignore it and use the default
}
// Connect to the database
Connection conn = DriverManager.getConnection (url, "scott", "tiger"); // Create a statement
Statement stmt = conn.createStatement ();
// Create the stored function
stmt.execute ("create or replace function RAISESAL (name CHAR, raise NUMBER)
return NUMBER is begin return raise + 100000; end;");
stmt.close(); // Close the statement
// Prepare to call the stored procedure RAISESAL. // This sample uses the SQL92 syntax
CallableStatement cstmt = conn.prepareCall ("{? = call RAISESAL (?, ?)}");
// Declare that the first ? is a return value of type Int
cstmt.registerOutParameter (1, Types.INTEGER); // We want to raise LESLIE's salary by 20,000
cstmt.setString (2, "LESLIE"); // The name argument is the second ?
cstmt.setInt (3, 20000); // The raise argument is the third ? // Do the raise
cstmt.execute ();
// Get the new salary back
int new_salary = cstmt.getInt (1);
System.out.println ("The new salary is: " + new_salary);
cstmt.close(); // Close the statement
conn.close(); // Close the connection
}
}

```

Self-Practice Program – 4

```

/***
* This sample shows how to use the batching extensions. In this example, we demonstrate the sue
*of the "sendBatch" API. This allows the user to actually execute a set of batched execute
*commands. */

// You need to import the java.sql package to use JDBC

import java.sql.*;
import oracle.jdbc.*;
class SendBatch
{
public static void main (String args []) throws SQLException
{
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
String url = "jdbc:oracle:oci8:@";

```

```

try {
String url1 = System.getProperty("JDBC_URL"); if(url1 != null)
url = url1;
} catch (Exception e) {
// If there is any security exception, ignore it and use the default
}
// Connect to the database
Connection conn = DriverManager.getConnection (url, "scott", "tiger");
Statement stmt = conn.createStatement ();
// Default batch value set to 50 for all prepared statements belonging to this
connection.((OracleConnection)conn).setDefaultExecuteBatch(50);
PreparedStatement ps = conn.prepareStatement ("insert into dept values (?, ?, ?
ps.setInt (1, 32);
ps.setString (2, "Oracle");
ps.setString (3, "USA");
// this execute does not actually happen at this point
System.out.println (ps.executeUpdate ());
ps.setInt (1, 33);
ps.setString (2, "Applications");
ps.setString (3, "Indonesia");
// this execute does not actually happen at this point
int rows = ps.executeUpdate ();
System.out.println ("Number of rows updated before calling sendBatch: " + rows);
// Execution of both previously batched executes will happen at this point.
//The number of rows updated will be returned by sendBatch.
rows = ((OraclePreparedStatement)ps).sendBatch ();
System.out.println ("Number of rows updated by calling sendBatch: " + rows);
ps.close ();
conn.close ();
}
}

```

Self-Practice Program – 5

```

/**
* This sample shows how to use the batching extensions.In this example, we set the defaultBatch
* value from the connection object. This affects all statements created from this connection. It is
* possible to set the batch value individually for each statement. The API to use on the statement
object is setExecuteBatch. * */

```

```

import java.sql.*;
import oracle.jdbc.*;

class SetExecuteBatch {
    public static void main (String args []) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
        String url = "jdbc:oracle:oci8:@";
        try {
            String url1 = System.getProperty("JDBC_URL");
            if(url1 != null)
                url = url1;
        } catch (Exception e) {
            // If there is any security exception, ignore it and use the default
        }
        // Connect to the database
        Connection conn = DriverManager.getConnection (url, "scott", "tiger");
        // Default batch value set to 2 for all prepared statements belonging to this
        connection>((OracleConnection)conn).setDefaultExecuteBatch (2);
        PreparedStatement ps = conn.prepareStatement ("insert into dept values (?, ?, ?)");
        ps.setInt (1, 12);
        ps.setString (2, "Oracle");
        ps.setString (3, "USA");
        // No data is sent to the database by this call to executeUpdate
        System.out.println ("Number of rows updated so far: " + ps.executeUpdate());
        ps.setInt (1, 11);
        ps.setString (2, "Applications");
        ps.setString (3, "Indonesia");
        // The number of batch calls to executeUpdate is now equal to the batch value of 2.
        // The data is now sent to the database and // both rows are inserted in a single roundtrip.
        int rows = ps.executeUpdate ();
        System.out.println ("Number of rows updated now: " + rows);
        ps.close ();
        conn.close();
    }
}

```

Introduction to HTML

What is HTML?

- HTML is a language for describing web pages.
- HTML stands for **Hyper Text Markup Language**
- HTML is not a programming language, it is a **markup language**
- A markup language is a set of **markup tags**
- The markup tags describe how text should be **displayed**

HTML Markup Tags

- HTML markup tags are usually called HTML tags
- HTML tags are keywords surrounded by angle brackets like <html>
- HTML tags normally come in pairs like and
- The first tag in a pair is the start tag, the second tag is the end tag
- The start and end tags are also called the opening and closing tags.

What is an HTML File?

- An HTML file is a text file with HTML tags
- An HTML file name must end with .htm or .html
- An HTML file can be created using a simple text editor
- An HTML file is often called an HTML document or a Web Page

When a browser displays a web page, it will not display the markup tags. The browser uses the markup tags to understand the layout of the page.

A very basic HTML Document

```
<html>
<body>
<p>This is my first paragraph</p>
<p>This is my <b>second</b> paragraph</p> </body>
</html>
```

Explanation:

- When a browser displays a web page, it will not display the markup tags.
- The text between the <html> and </html> tags describes a web page.
- The text between the <body> and </body> tags is displayed in the web browser.
- The text between the <p> and </p> tags is displayed as paragraphs.
- The text between the and tags is displayed in a bold font.

Writing HTML

Generally, you'll use a plain text editor (like Notepad) to edit HTML. This is a good way to learn HTML. However, professional web developers often prefer HTML editors like FrontPage or Dreamweaver, instead of writing plain text.

Basic HTML Tags

- **HTML Headings**

Headings are defined with the `<h1>` to `<h6>` tags. `<h1>` defines the largest heading. `<h6>` defines the smallest heading.

```
<h1>This is a heading</h1>
<h2>This is a heading</h2>
<h3>This is a heading</h3>
```

Result:

This is a heading

This is a heading

This is a heading

Note: HTML automatically displays an empty line before and after headings.

- **HTML Paragraphs**

Paragraphs are defined with the `<p>` tag.

```
<p>This is a paragraph</p>
<p>This is another paragraph</p>
```

Note: HTML automatically displays an empty line before and after a paragraph. Don't Forget the End Tag

Most browsers will display HTML correctly even if you forget the end tag; for example

```
<p>This is a paragraph
<p>This is another paragraph
```

This example will work in most browsers, but don't rely on it. Forgetting the end tag can produce unexpected results or errors.

• HTML Line Breaks

Use the
 tag if you want a line break (a new line) without starting a new paragraph:

```
<p>This is<br>a para<br>graph with line breaks</p>
```

Note: The
 tag is an empty tag. It has no end tag like </br>.

• HTML Comments

Comments can be inserted in the HTML code to make it more readable and understandable. Comments are ignored by the browser and not displayed. Comments are written like this:

```
<!-- This is a comment -->
```

Note: There is an exclamation point after the opening bracket, but not before the closing bracket.

- With HTML, you cannot change the output by adding extra spaces or extra lines in your HTML code. The browser will remove extra spaces and extra lines when the page is displayed. Any number of lines counts as one space, and any number of spaces count as one space.

HTML Tag Convention:

HTML tags are not case sensitive: <P> means the same as <p>. Plenty of web sites use uppercase HTML tags in their pages. However it's recommended to use lowercase tags because the World Wide Web Consortium (W3C) recommends lowercase in HTML 4, and demands lowercase tags in newer versions of (X)HTML.

HTML Attributes

- HTML elements can have attributes
- Attributes provide additional information about the element
- Attributes are always specified in the start tag

Attribute Syntax

Attributes always come in name/value pairs like this: name="value".

Attributes Example 1:

<h1> defines the start of a heading.

```
<h1 align="center">
```

has additional information about the alignment.

Attributes Example 2:

<body> defines the body of an HTML document.

```
<body bgcolor="yellow">
```

has additional information about the background color.

Attributes Example 3:

<table> defines an HTML table. (You will learn more about HTML tables later)

```
<table border="1">
```

has additional information about the border around the table.

- **Use Lowercase Attributes**

Attributes and attribute values are case-insensitive. However, the World Wide Web Consortium (W3C) recommends lowercase attributes/attribute values in their HTML 4 recommendation, and XHTML demands lowercase attributes/attribute values.

- **Always Quote Attribute Values**

Attribute values should always be enclosed in quotes. Double style quotes are the most common, but single style quotes are also allowed. In some rare situations, like when the attribute value itself contains quotes, it is necessary to use single quotes:

```
name='John "ShotGun" Nelson'
```

HTML Text Formatting

This text is bold

This text is big

This text is italic

This is computer output

This is _{subscript} and ^{superscript}

HTML Formatting Tags

HTML uses tags like and <i> for formatting output, like bold or italic text. These HTML tags are called formatting tags.

Text Formatting Tags

<u>Tag</u>	<u>Description</u>
	Defines bold text
<big>	Defines big text
	Defines emphasized text
<i>	Defines italic text

<code><small></code>	Defines small text
<code></code>	Defines strong text
<code><sub></code>	Defines subscripted text
<code><sup></code>	Defines superscripted text
<code><ins></code>	Defines inserted text
<code></code>	Defines deleted text
<code><s></code>	Deprecated. Use <code></code> instead
<code><strike></code>	Deprecated. Use <code></code> instead
<code><u></code>	Deprecated. Use styles instead

Computer Output Tags

<u>Tag</u>	<u>Description</u>
<code><code></code>	Defines computer code text
<code><kbd></code>	Defines keyboard text
<code><samp></code>	Defines sample computer code
<code><tt></code>	Defines teletype text
<code><var></code>	Defines a variable
<code><pre></code>	Defines preformatted text
<code><listing></code>	Deprecated. Use <code><pre></code> instead
<code><plaintext></code>	Deprecated. Use <code><pre></code> instead
<code><xmp></code>	Deprecated. Use <code><pre></code> instead

Citations, Quotations, and Definition Tags

<u>Tag</u>	<u>Description</u>
<code><abbr></code>	Defines an abbreviation
<code><acronym></code>	Defines an acronym
<code><address></code>	Defines an address element
<code><bdo></code>	Defines the text direction
<code><blockquote></code>	Defines a long quotation
<code><q></code>	Defines a short quotation
<code><cite></code>	Defines a citation
<code><dfn></code>	Defines a definition term

HTML Character Entities

Some characters are reserved in HTML. For example, you cannot use the greater than or less than signs within your text because the browser could mistake them for markup.

If we want the browser to actually display these characters we must insert character entities in the HTML source.

A character entity looks like this: &entity_name; OR &#entity_number;
To display a less than sign we must write: < or <

The advantage of using an entity name instead of a number is that the name often is easier to remember. However, the disadvantage is that browsers may not support all entity names (while the support for entity numbers is very good).

Non-breaking Space

The most common character entity in HTML is the non-breaking space. Normally HTML will truncate spaces in your text. If you write 10 spaces in your text HTML will remove 9 of them. To add lots of spaces to your text, use the character entity.

Commonly Used Character Entities

Note: Entity names are case sensitive!

<u>Result</u>	<u>Description</u>	<u>Entity Name</u>	<u>Entity Number</u>
<	non-breaking space	 	
>	less than	<	<
>	greater than	>	>
&	ampersand	&	&
₵	cent	¢	¢
£	pound	£	£
¥	yen	¥	¥
€	euro	€	€
©	copyright	©	©
®	registered trademark	®	®

HTML Links

HTML uses a hyperlink to link to another document on the Web.

The Anchor Tag and the Href Attribute

HTML uses the <a> (anchor) tag to create a link to another document.

An anchor can point to any resource on the Web: an HTML page, an image, a sound file, a movie, etc. The syntax of creating an anchor:

Text to be displayed

The <a> tag is used to create an anchor to link from, the href attribute is used to address the document to link to, and the words between the open and close of the anchor tag will be displayed as a hyperlink. This anchor defines a link to google.com

Visit to Google Search Engine!

The Target Attribute

With the target attribute, you can define where the linked document will be opened. The line below will open the document in a new browser window:

```
<a href="http://www.hpesindia.com/" target="_blank">Visit hpes website!</a>
```

The Anchor Tag and the Name Attribute

The name attribute is used to create a named anchor. When using named anchors we can create links that can jump directly into a specific section on a page, instead of letting the user scroll around to find what he/she is looking for.

Below is the syntax of a named anchor:

```
<a name="label">Text to be displayed</a>
```

The name attribute is used to create a named anchor. The name of the anchor can be any text you care to use. The line below defines a named anchor:

```
<a name="tips">Useful Tips Section</a>
```

You should notice that a named anchor is not displayed in a special way.

To link directly to the "tips" section, add a # sign and the name of the anchor to the end of a URL, like this:

```
<a href="http://www.javatutorials.com/chapter1.jsp#tips">Jump to the Useful Tips Section</a>
```

A hyperlink to the Useful Tips Section from WITHIN the file "chapter1.jsp" will look like this:

```
<a href="#tips">Jump to the Useful Tips Section</a>
```

Useful Tips

- Always add a trailing slash to subfolder references. If you link like this:

href="http://www.hpesindia.com/html", you will generate two HTTP requests to the server, because the server will add a slash to the address and create a new request like this:

```
href="http://www.hpesindia.com/html/"
```

- Named anchors are often used to create "table of contents" at the beginning of a large document. Each chapter within the document is given a named anchor, and links to each of these anchors are put at the top of the document.

- If a browser cannot find a named anchor that has been specified, it goes to the top of the document. No error occurs.

HTML Frames

With frames, you can display more than one HTML document in the same browser window. Each HTML document is called a frame, and each frame is independent of the others.

The disadvantages of using frames are:

- The web developer must keep track of more HTML documents
- It is difficult to print the entire page

The Frameset Tag

- The `<frameset>` tag defines how to divide the window into frames
- Each frameset defines a set of rows or columns
- The values of the rows/columns indicate the amount of screen area each row/column will occupy

The Frame Tag

- The `<frame>` tag defines what HTML document to put into each frame

In the example below we have a frameset with two columns. The first column is set to 25% of the width of the browser window. The second column is set to 75% of the width of the browser window. The HTML document "frame_a.htm" is put into the first column, and the HTML document "frame_b.htm" is put into the second column:

```
<frameset cols="25%,75%">
  <frame src="frame_a.htm">
  <frame src="frame_b.htm">
</frameset>
```

Note: The frameset column size value can also be set in pixels (`cols="200,500"`), and one of the columns can be set to use the remaining space (`cols="25%,*"`).

Useful Tips

If a frame has visible borders, the user can resize it by dragging the border. To prevent a user from doing this, you can add `noresize="noresize"` to the `<frame>` tag.

Add the `<noframes>` tag for browsers that do not support frames.

Important: You cannot use the `<body></body>` tags together with the `<frameset></frameset>` tags! However, if you add a `<noframes>` tag containing some text for browsers that do not support frames, you will have to enclose the text in `<body></body>` tags!

Frame Tags

<u>Tag</u>	<u>Description</u>
<frameset>	Defines a set of frames
<frame>	Defines a sub window (a frame)
<noframes>	Defines a no frame section for browsers that do not handle frames
<iframe>	Defines an inline sub window (frame)

HTML Tables

Tables are defined with the <table> tag. A table is divided into rows (with the <tr> tag), and each row is divided into data cells (with the <td> tag). The letters td stands for "table data," which is the content of a data cell. A data cell can contain text, images, lists, paragraphs, forms, horizontal rules, tables, etc.

```
<table border="1">
  <tr>
    <td>row 1, cell 1</td>
    <td>row 1, cell 2</td>
  </tr>
  <tr>
    <td>row 2, cell 1</td>
    <td>row 2, cell 2</td>
  </tr>
</table>
```

Here is how it'll appear in the browser window

row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

Tables and the Border Attribute

If you do not specify a border attribute the table will be displayed without any borders. Sometimes this can be useful, but most of the time, you want the borders to show. To display a table with borders, you will have to use the border attribute:

```
<table border="1">
  <tr>
    <td>Row 1, cell 1</td>
    <td>Row 1, cell 2</td>
  </tr>
</table>
```

Headings in a Table

Headings in a table are defined with the `<th>` tag.

```
<table border="1">
<tr>
    <th>Heading</th>  <th>Another Heading</th>
</tr>
<tr>
    <td>row 1, cell 1</td>      <td>row 1, cell 2</td>
</tr>
<tr>
    <td>row 2, cell 1</td> <td>row 2, cell 2</td>
</tr>
</table>
```

How it looks in a browser:

Heading	Another Heading
row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

Table Tags

<u>Tag</u>	<u>Description</u>
<table>	Defines a table
<th>	Defines a table header
<tr>	Defines a table row
<td>	Defines a table cell
<caption>	Defines a table caption
<colgroup>	Defines groups of table columns
<col>	Defines the attribute values for one or more columns in a table
<thead>	Defines a table head
<tbody>	Defines a table body
<tfoot>	Defines a table footer

HTML Lists

HTML supports ordered, unordered and definition lists.

Unordered Lists

An unordered list is a list of items. The list items are marked with bullets (typically small black circles). An unordered list starts with the `` tag. Each list item starts with the `` tag.

```
<ul>
    <li>Coffee</li>
    <li>Milk</li>
</ul>
```

Here is how it looks in a browser:

- Coffee
- Milk

Inside a list item you can put paragraphs, line breaks, images, links, other lists, etc.

Ordered Lists

An ordered list is also a list of items. The list items are marked with numbers. An ordered list starts with the `` tag. Each list item starts with the `` tag.

```
<ol>
    <li>Coffee</li>
    <li>Milk</li>
</ol>
```

Here is how it looks in a browser:

1. Coffee
2. Milk

Inside a list item you can put paragraphs, line breaks, images, links, other lists, etc

Definition Lists

A definition list is not a list of items. This is a list of terms and explanation of the terms. A definition list starts with the `<dl>` tag. Each definition-list term starts with the `<dt>` tag. Each definition-list definition starts with the `<dd>` tag.

```
<dl>
    <dt>Coffee</dt>
    <dd>Black hot drink</dd>
    <dt>Milk</dt>
    <dd>White cold drink</dd>
</dl>
```

Here is how it looks in a browser:

Coffee	Black hot drink
Milk	White cold drink

Inside a definition-list definition (the `<dd>` tag) you can put paragraphs, line breaks, images, links, other lists, etc.

List Tags

<u>Tag</u>	<u>Description</u>
	Defines an ordered list
	Defines an unordered list
	Defines a list item
<dl>	Defines a definition list
<dt>	Defines a definition term
<dd>	Defines a definition description
<dir>	Deprecated. Use instead
<menu>	Deprecated. Use instead

HTML Forms and Input

Forms

A form is an area that can contain form elements.

Form elements are elements that allow the user to enter information (like text fields, textarea fields, drop-down menus, radio buttons, checkboxes, etc.) in a form.

A form is defined with the <form> tag.

```
<form>
    <input>
    <input>
</form>
```

Input

The most used form tag is the <input> tag. The type of input is specified with the type attribute. The most commonly used input types are explained below.

Text Fields

Text fields are used when you want the user to type letters, numbers, etc. in a form.

```
<form>
First name: <input type="text" name="firstname">
<br>
Last name: <input type="text" name="lastname">
</form>
```

How it looks in a browser:

First name :

Last name :

Note that the form itself is not visible. Also note that in most browsers, the width of the text field is 20 characters by default.

Radio Buttons

Radio Buttons are used when you want the user to select one of a limited number of choices.

```
<form>
  <input type="radio" name="sex" value="male"> Male <br>
  <input type="radio" name="sex" value="female"> Female
</form>
```

How it looks in a browser:

- Male
- Female

Note that only one option can be chosen.

Checkboxes

Checkboxes are used when you want the user to select one or more options of a limited number of choices.

```
<form>
  I have a bike: <input type="checkbox" name="vehicle" value="Bike"> <br>
  I have a car: <input type="checkbox" name="vehicle" value="Car"> <br>
  I have an airplane: <input type="checkbox" name="vehicle" value="Airplane">
</form>
```

How it looks in a browser:

- I have a bike:
- I have a car:
- I have an airplane:

The Form's Action Attribute and the Submit Button

When the user clicks on the "Submit" button, the content of the form is sent to the server. The form's action attribute defines the name of the file to send the content to. The file defined in the action attribute usually does something with the received input.

```
<form name="input" action="html_form_submit.jsp" method="get">
  Username: <input type="text" name="user">
  <input type="submit" value="Submit">
</form>
```

If you type some characters in the text field above, and click the "Submit" button, the browser will send your input to a page called "html_form_submit.jsp". The page will show you the received input.

Form Tags

<u>Tag</u>	<u>Description</u>
<form>	Defines a form for user input
<input>	Defines an input field
<textarea>	Defines a text-area (a multi-line text input control)
<label>	Defines a label to a control
<fieldset>	Defines a fieldset
<legend>	Defines a caption for a fieldset
<select>	Defines a selectable list (a drop-down box)
<optgroup>	Defines an option group
<option>	Defines an option in the drop-down box
<button>	Defines a push button
<isindex>	Deprecated. Use <input> instead

HTML Images

With HTML you can display images in a document.

The Image Tag and the Src Attribute

In HTML, images are defined with the tag.

The tag is empty, which means that it contains attributes only and it has no closing tag. To display an image on a page, you need to use the src attribute. Src stands for "source". The value of the src attribute is the URL of the image you want to display on your page.

The syntax of defining an image:

```

```

The URL points to the location where the image is stored. An image named "hplogo.gif" located in the directory "images" on "www.hpesindia.com" has the URL:
<http://www.hpesindia.com/images/hplogo.gif>.

The browser puts the image where the image tag occurs in the document. If you put an image tag between two paragraphs, the browser shows the first paragraph, then the image, and then the second paragraph.

The Alt Attribute

The alt attribute is used to define an "alternate text" for an image. The value of the alt attribute is an author-defined text:

```

```

The "alt" attribute tells the reader what he or she is missing on a page if the browser can't load images. The browser will then display the alternate text instead of the image. It is a good practice to

include the "alt" attribute for each image on a page, to improve the display and usefulness of your document for people who have text-only browsers.

Image Tags

<u>Tag</u>	<u>Description</u>
	Defines an image
<map>	Defines an image map
<area>	Defines a clickable area inside an image map

HTML Backgrounds

A good background can make a Web site look really great.

Backgrounds

The <body> tag has two attributes where you can specify backgrounds. The background can be a color or an image.

Bgcolor

The bgcolor attribute specifies a background-color for an HTML page. The value of this attribute can be a hexadecimal number, an RGB value, or a color name:

```
<body bgcolor="#000000">  
<body bgcolor="rgb(0,0,0)">  
<body bgcolor="black">
```

The lines above all set the background-color to black.

Background

The background attribute specifies a background-image for an HTML page. The value of this attribute is the URL of the image you want to use. If the image is smaller than the browser window, the image will repeat itself until it fills the entire browser window.

```
<body background="VirtualBalls.gif">  
<body background="http://www.hpesindia.com/VirtualBalls.gif">
```

The URL can be relative (as in the first line above) or absolute (as in the second line above).

Note: If you want to use a background image, you should keep in mind:

- Will the background image increase the loading time too much?
- Will the background image look good with other images on the page?
- Will the background image look good with the text colors on the page?
- Will the background image look good when it is repeated on the page?
- Will the background image take away the focus from the text?

Useful Tips

Style sheets (CSS) should be used instead (to define the layout and display properties of HTML elements).

Web Application & JAVA Servlet Technology

Web Applications

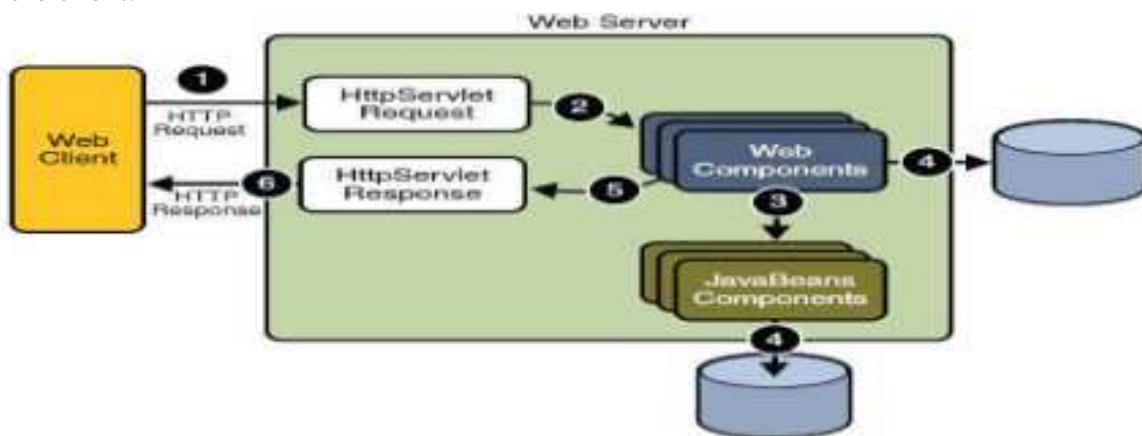
A web application is a dynamic extension of a web or application server. There are two types of web applications:

- **Presentation-oriented:** A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XML, and so on) and dynamic content in response to requests.
- **Service-oriented:** A service-oriented web application implements the endpoint of a web service. Presentation-oriented applications are often clients of service-oriented web applications.

Note: Service-oriented web applications are not in the scope of this training.

Web Components

In the Java 2 platform, web components provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in the following figure. The client sends an HTTP request to the web server. A web server that implements Java Servlet and Java Server Pages technology converts the request into an HttpServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HttpServletResponse object or it can pass the request to another web component. Eventually a web component generates a HttpServletResponse object. The web server converts this object to an HTTP response and returns it to the client.



Role of Servlets & JSPs (A Quick View)

Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service en are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), and XML.

Web Container & DD (A Quick View)

Web components are supported by the services of a runtime platform called a web container. A web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email. Certain aspects of web application behaviour can be configured when the application is installed, or deployed, to the web container. The configuration information is maintained in a text file in XML format called a web application deployment descriptor (DD). A DD must conform to the schema described in the Java Servlet Specification.

Benefits of Servlets

In the beginning, Internet consisted of only static contents written using Hypertext Markup Language (HTML). At that time, anyone who could author HTML pages was considered an Internet expert.

This did not last long, however.

Soon dynamic web contents were made possible through the Common Gateway Interface (CGI) technology. CGI enables the web server to call an external program and pass HTTP request information to that external program to process the request. The response from the external program is then passed back to the web server, which forwards it to the client browser. Over the course of time, Perl became the most popular language to write CGI programs.

As the Internet became more and more popular, however, the number of users visiting a popular web site increased exponentially, and it became apparent that CGI had failed to deliver scalable Internet applications. The flaw in CGI is that each client request makes the web server spawn a new process of the requested CGI program. As we all know, process creation is an expensive operation that consumes a lot of CPU cycles and computer memory.

Gradually, new and better technologies replaced CGI as the main technology for web application development. The JAVA Servlet Technology is one of these, introduced by Sun Microsystems in 1996. JSP is just an extension of servlet technology.

Servlet (and JSP) offers the following benefits:

- **Performance:** The performance of servlets is superior to CGI because there is no process creation for each client request. Instead, each request is handled by the servlet container process. After a servlet is finished processing a request, it stays resident in memory, waiting for another request.
- **Portability:** Similar to other Java technologies, servlet applications are portable. You can move them to other operating systems without serious hassles.
- **Rapid development cycle:** As a Java technology, servlets have access to the rich Java library, which helps speed up the development process.
- **Robustness:** Servlets are managed by the Java Virtual Machine. As such, you don't need to worry about memory leak or garbage collection, which helps you write robust applications.
- **Widespread acceptance:** Java is a widely accepted technology. This means that numerous vendors work on Java-based technologies. One of the advantages of this widespread acceptance is that you can easily find and purchase components that suit your needs, which saves precious development time.

Servlet Application Architecture

A servlet is a Java class that can be loaded dynamically into and run by a special web server. This servlet-aware web server is called a servlet container, which also was called a servlet engine in the early days of the servlet technology.

Servlets interact with clients via a request-response model based on HTTP. Because servlet technology works on top of HTTP, a servlet container must support HTTP as the protocol for client requests and server responses. However, a servlet container also can support similar protocols, such as HTTPS (HTTP over SSL) for secure transactions. Following figure provides the architecture of a servlet application.

The servlet application architecture.



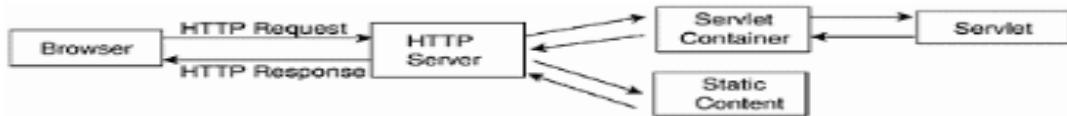
In a JSP application, the servlet container is replaced by a JSP container. Both the servlet container and the JSP container often are referred to as the web container or servlet/JSP container, especially if a web application consists of both servlets and JSP pages.

Note: You will learn more about servlet and JSP containers further.

As you can see in the above figure, a servlet application also can include static content, such as HTML pages and image files. Allowing the servlet container to serve static content is not preferable because the content is faster if served by a more robust HTTP server, such as the Apache web server or Microsoft Internet Information Server. As such, it is common practice to put a web server at the front to handle all client requests. The web server serves static content and passes to the

servlet containers all client requests for servlets. Following Figure shows a more common architecture for a servlet application.

The servlet application architecture employing an HTTP server.

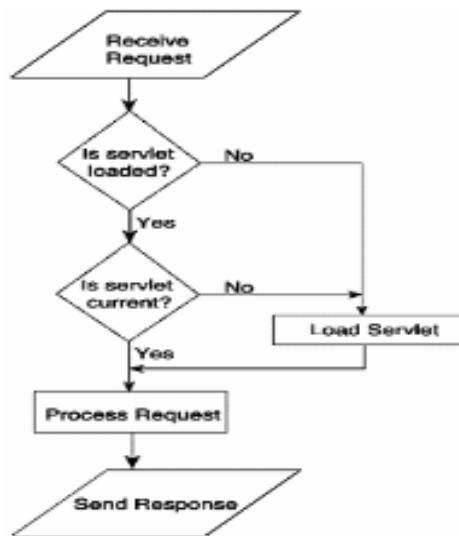


Note: A Java web application architecture employing a J2EE server is different from the above diagrams. You'll learn it further.

How a Servlet Works

A servlet is loaded by the servlet container the first time the servlet is requested. The servlet then is forwarded the user request, processes it, and returns the response to the servlet container, which in turn sends the response back to the user. After that, the servlet stays in memory waiting for other requests—it will not be unloaded from the memory unless the servlet container sees a shortage of memory. Each time the servlet is requested, however, the servlet container compares the timestamp of the loaded servlet with the servlet class file. If the class file timestamp is more recent, the servlet is reloaded into memory. This way, you don't need to restart the servlet container every time you update your servlet.

The way in which a servlet works inside the servlet container is depicted in the following diagram.



The Tomcat Servlet Container

A number of servlet containers are available today. The most popular one—and the one recognized as the official servlet/JSP container—is Tomcat. Originally designed by Sun Microsystems, Tomcat source code was handed over to the Apache Software Foundation in October 1999. In this new

home, Tomcat was included as part of the Jakarta Project, one of the projects of the Apache Software Foundation. Working through the Apache process, Apache, Sun, and other companies—with the help of volunteer programmers worldwide—turned Tomcat into a world-class servlet reference implementation.

Tomcat by itself is a web server. This means that you can use Tomcat to service HTTP requests for servlets, as well as static files (HTML, image files, and so on). In practice, however, since it is faster for non-servlet, non-JSP requests, Tomcat normally is used as a module with another more robust web server, such as Apache web server or Microsoft Internet Information Server. Only requests for servlets or JSP pages are passed to Tomcat.

To write a servlet, you need at least version 1.2 of the Java Development Kit. The reference implementation for both servlets and JSP are not included in J2SE, but they are included in Tomcat. Tomcat is written purely in Java.

Six Steps to Running Your First Servlet

After you have installed and configured Tomcat, you can put it into service. Basically, you need to follow six steps to go from writing your servlet to running it. These steps are summarized as follows:

1. Create a directory structure under Tomcat for your application.
2. Write the servlet source code. You need to import the javax.servlet package and the javax.servlet.http package in your source file.
3. Compile your source code.
4. Create a deployment descriptor.
5. Run Tomcat.
6. Call your servlet from a web browser.

Step 1: Create a Directory Structure Under Tomcat

Note:-

The directory where Tomcat is installed is often referred to as %CATALINA_HOME%. In previous versions of Tomcat, this directory was called %TOMCAT_HOME%.

When you install Tomcat, several subdirectories are automatically created under the Tomcat home directory (%CATALINA_HOME%). One of the subdirectories is webapps. The webapps directory is where you store your web applications. A web application is a collection of servlets and other content installed under a specific subset of the server's URL namespace. A separate directory is dedicated for each servlet application. Therefore, the first thing to do when you build a servlet application is create an application directory. To create a directory structure for an application called myApp, follow these steps:

1. Create a directory called myApp under the webapps directory. The directory name is important because this also appears in the URL to your servlet.

2. Create a directory with name “WEB-INF” under myApp, and create a directory named classes under WEB-INF. The directory structure is shown in the following figure. The directory classes under WEB-INF is for your Java classes. If you have HTML files, put them directly under the myApp directory. You may also want to create a directory called images under myApp for all your image files.

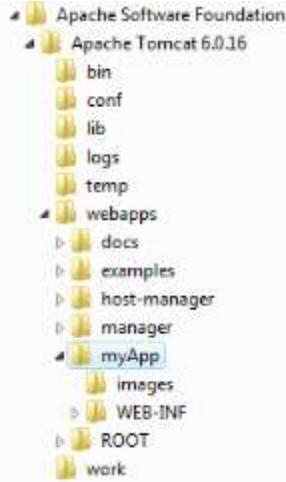


Figure: Tomcat application directory structure.

Note: You'll find other directories such as examples, manager, ROOT, tomcat-doc, and webdav in webapps. These directories are for applications that are created automatically when you install Tomcat.

Step 2: Write the Servlet Source Code

Now, it's time to write the servlet's source code. You can use any text editor for this purpose.

The code given below is a simple servlet called TestingServlet. The file is named TestingServlet.java. The servlet sends a few HTML tags and some text to the browser. For now, don't worry if you haven't got a clue about how it works.

TestingServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class TestingServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Servlet Testing</TITLE>");
```

```
out.println("</HEAD>");  
out.println("<BODY>");  
out.println("Welcome to the Servlet Testing Center");  
out.println("</BODY>");  
out.println("</HTML>");  
}  
}
```

Now, save your TestingServlet.java file to the WEB-INF/classes directory under myApp. Placing your source code here will make it inaccessible from a web browser. Static files, such as HTML files and image files, should be placed directly under the myApp directory or a directory under it.

Warning: Placing your source code files outside the WEB-INF directory will make them viewable from a browser.

Step 3: Compile Your Source Code

For your servlet source code to compile, you need to include the path to the servlet.jar file in your CLASSPATH environment variable. The servlet.jar is located in the common\lib\ subdirectory under %CATALINA_HOME%.

For example, if you installed Tomcat under the C:\drive on Windows and you named the install directory tomcat, type the following command from the directory where TestingServlet.java resides.

```
javac classpath C:\tomcat\common\lib\servlet.jar TestingServlet.java
```

Alternatively, to save you typing the class path every time you compile your source code, you can add the complete path to the servlet.jar file to your CLASSPATH environment variable. Again, if you have installed Tomcat under C:\ and named the install directory tomcat, you must add C:\tomcat\ common\lib\servlet.jar to the CLASSPATH environment variable. Afterward, you can compile your source by simply typing the following.

```
javac TestingServlet.java
```

Note: If you are using Windows, remember that the new environment variable takes effect only for new console windows. In other words, after changing a new environment variable, open a new console window for typing in your command lines.

Step 4: Create the Deployment Descriptor

A deployment descriptor is an optional component in a servlet application. The descriptor takes the form of an XML document called web.xml and must be located in the WEB-INF directory of the servlet application. When present, the deployment descriptor contains configuration settings specific to that application. Deployment descriptors are discussed in detail further.

To create the deployment descriptor, you now need to create a web.xml file and place it under the WEB-INF directory under myApp.

The web.xml for this example application must have the following content.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<servlet>
<servlet-name>Testing</servlet-name>
<servlet-class>TestingServlet</servlet-class>  </servlet>
</web-app>
```

The web.xml file has one element—web-app. You should write all your servlets under <web-app>. For each servlet, you have a <servlet> element and you need the <servlet-name> and <servlet-class> elements. The <servlet-name> is the name for your servlet, by which it is known to Tomcat. The <servlet-class> is the compiled file of your servlet without the .class extension.

Having more than one servlet in an application is very common. For every servlet, you need a <servlet> element in the web.xml file. For example, the following shows you how web.xml looks if you add another servlet called Login:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<servlet>
<servlet-name>Testing</servlet-name>
<servlet-class>TestingServlet</servlet-class>
</servlet>

<servlet>
<servlet-name>Login</servlet-name>
<servlet-class>LoginServlet</servlet-class>
</servlet>
</web-app>
```

Step 5: Run Tomcat

If Tomcat is not already running, you need to start it. Ask your trainer how to run Tomcat. Step 6: Call Your Servlet from a Web Browser

Now, you can call your servlet from a web browser. By default, Tomcat runs on port 8080 in the myApp virtual directory under the servlet subdirectory. The servlet that you wrote in the preceding steps is named Testing. The URL for that servlet has the following format: `http://domain-name/virtual-directory/servlet/servlet-name`

Any static file can be accessed using the following URL:

`http://domain-name/virtual-directory/staticFile.html`

For example, a Logo.gif file under the myApp/images/ directory can be accessed using the following URL.

`http://domain-name/virtual-directory/images/Logo.gif`

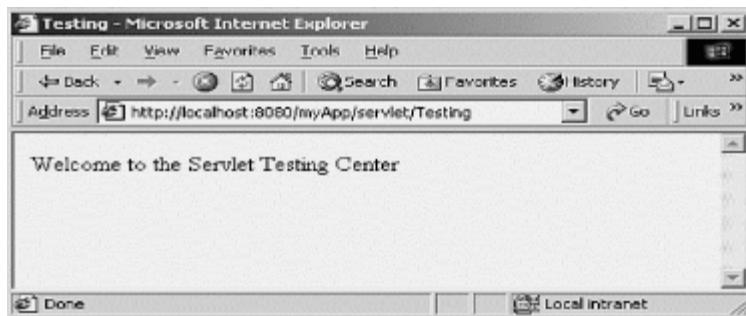
If you run the web browser from the same computer as Tomcat, you can replace the domain-name part with "localhost". In that case, the URL for your servlet is

`http://localhost:8080/myApp/servlet/Testing`

In the deployment descriptor you wrote in Step 4, you actually mapped the servlet class file called TestingServlet with the name "Testing," so that your servlet can be called by specifying its class file (TestingServlet) or its name (Testing). Without a deployment descriptor, you must call the servlet by specifying its class name; that is, TestingServlet. This means that if you did not write a deployment descriptor in Step 4, you need to use the following URL to call your servlet:

`http://localhost:8080/myApp/servlet/TestingServlet`

Typing the URL in the Address or Location box of your web browser will give you the string "Welcome to the Servlet Testing Center," as shown in the following figure.



Congrats! You have just written your first servlet.

Note: 8080 is the default port no. of Tomcat. However, you can change it. Ask your trainer how to do so.

Using NetBeans IDE to develop Web Applications

Till now, you developed the web application manually. You followed the above mentioned 6 steps yourself. It's very cumbersome, especially when the application is very large. Most often, you would like to use an IDE for fast, easy and error-free development. NetBeans IDE is one of the best suitable IDEs for such development.

Inside Servlets

In the previous chapter, you learnt how to develop a web application. To be an expert, however, you need to understand the nuts and bolts of the Java Servlet specification Application Programming Interface (API).

Two packages are available for servlet programmers: javax.servlet and javax.servlet.http. The first one contains basic classes and interfaces that you can use to write servlets from the scratch. The second package, javax.servlet.http, offers more advanced classes and interfaces that extend classes and interfaces from the first package. It is much more convenient to program using the second package.

The javax.servlet Package

The javax.servlet package contains seven interfaces, three classes, and two exceptions.

The seven interfaces are as follows:

- RequestDispatcher
- Servlet
- ServletConfig
- ServletContext
- ServletRequest
- ServletResponse
- SingleThreadModel

The three classes are as follows:

- GenericServlet
- ServletInputStream
- ServletOutputStream

And, finally, the exception classes are these:

- ServletException
- UnavailableException

A Servlet's Life Cycle

The javax.servlet.Servlet interface in the javax.servlet package is the source of all activities in servlet programming. Servlet is the central abstraction of the Java servlet technology. Every servlet

you write must implement this javax.servlet.Servlet interface, either directly or indirectly. The life cycle of a servlet is determined by three of its methods: init, service, and destroy.

The init() Method

The init method is called by the servlet container after the servlet class has been instantiated. The servlet container calls this method exactly once to indicate to the servlet that the servlet is being placed into service. The init method must complete successfully before the servlet can receive any requests.

You can override this method to write initialization code that needs to run only once, such as loading a database driver, initializing values, and so on. In other cases, you normally leave this method blank.

The signature of this method is as follows:

```
public void init(ServletConfig config) throws ServletException
```

The init method is important also because the servlet container passes a ServletConfig object, which contains the configuration values stated in the web.xml file for this application. You'll find more on ServletConfig later in this chapter.

This method also can throw a ServletException. The servlet container cannot place the servlet into service if the init method throws a ServletException or the method does not return within a time period defined by the web server.

Note: ServletException is the most important exception in servlet programming. In fact, a lot of methods in the javax.servlet and javax.servlet.http packages can throw this exception when a problem exists in a servlet.

The service() Method

The service method is called by the servlet container after the servlet's init method to allow the servlet to respond to a request.

Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently. Therefore, you must be aware to synchronize access to any shared resources, such as files, network connections, and the servlet's class and instance variables. For example, if you open a file and write to that file from a servlet, you need to remember that a different thread of the same servlet also can open the same file. You'll how to create thread-safe servlets further.

This method has the following signature:

```
public void service(ServletRequest request, ServletResponse response)  
throws ServletException, java.io.IOException
```

The servlet container passes a ServletRequest object and the ServletResponse object. The ServletRequest object contains the client's request and the ServletResponse contains the servlet's

response. These two objects are important because they enable you to write custom code that determines how the servlet services the client request.

The service method throws a ServletException if an exception occurs that interferes with the servlet's normal operation. The service method also can throw a java.io.IOException if an input or output exception occurs during the execution of this method.

As the name implies, the service method exists so that you can write code that makes the servlet function the way it is supposed to.

The destroy() Method

The servlet container calls the destroy method before removing a servlet instance from service. This normally happens when the servlet container is shut down or the servlet container needs some free memory.

This method is called only after all threads within the servlet's service method have exited or after a timeout period has passed. After the servlet container calls this method, it will not call the service method again on this servlet.

The destroy method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, and threads) and make sure that any persistent state is synchronized with the servlet's current state in memory.

The signature of this method is as follows:

```
public void destroy()
```

Demonstrating the Life Cycle of a Servlet

The following example contains the code for a servlet named PrimitiveServlet, a very simple servlet that exists to demonstrate the life cycle of a servlet. The PrimitiveServlet class implements javax.servlet.Servlet (as all servlets must) and provides implementations for all the five methods of servlet. What it does is very simple. Each time any of the init, service, or destroy methods is called, the servlet writes the method's name to the console.

PrimitiveServlet.java

```
import javax.servlet.*;
import java.io.IOException;

public class PrimitiveServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
        System.out.println("init");
    }

    public void service(ServletRequest request, ServletResponse response) throws ServletException,
    IOException {
```

```

        System.out.println("service");
    }
    public void destroy() {
        System.out.println("destroy");
    }

}

public String getServletInfo() {
    return null;
}
public ServletConfig getServletConfig() {
    return null;
}
}

```

After you compile the source code into the myApp\WEB-INF\classes directory, add the servlet to the web.xml under the name Primitive, as shown below.

The web.xml File for PrimitiveServlet

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>Primitive</servlet-name>
        <servlet-class>PrimitiveServlet</servlet-class>    </servlet>
    </web-app>

```

You should then be able to call this servlet from your browser by typing the following URL:

http://localhost:8080/myApp/servlet/Primitive

The first time the servlet is called, the console displays these two lines:

init
service

This tells you that the init method is called, followed by the service method. However, on subsequent requests, only the service method is called. The servlet adds the following line to the console:

service

This proves that the init method is called only once. What are the getServletInfo and getServletConfig doing in the above example code. Nothing. They can be useful, but in the PrimitiveServlet class, they are just there to meet the specification that a class must provide implementations for all methods in the interface it implements.

You can return any string in the getServletInfo method, such as your company name or the author name or other information deemed necessary. Other people might extend your servlet class and might want to know what useful information the designer of the servlet has provided.

The getServletConfig is more important. We will see how it can be of use next.

Obtaining Configuration Information

The servlet specification allows you to configure your application. You'll find more information on this topic in the discussion of the deployment descriptor further. In this chapter, An example that demonstrates how you can retrieve configuration information from the application web.xml file, is presented.

For each servlet registered in the web.xml file, you have the option of specifying a set of initial parameter name/value pairs that you can retrieve from inside the servlet. The following web.xml file contains a servlet called ConfigDemo whose class is named ConfigDemoServlet.class. The servlet has two initial parameter name/value pairs. The first parameter is named adminEmail and its value is tom@abc.com. The second parameter is named adminContactNumber and the value for this parameter is 09005017996.

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE web-app  
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"  
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">  
<web-app>  
  <servlet>  
    <servlet-name>ConfigDemo</servlet-name>  
    <servlet-class>ConfigDemoServlet</servlet-class>  
    <init-param>  
      <param-name>adminEmail</param-name>  
      <param-value>tom@abc.com</param-value>  
    </init-param>  
    <init-param>  
      <param-name>adminContactNumber</param-name>  
      <param-value>09005017996</param-value>  
    </init-param>  
  </servlet>  
</web-app>
```

Why would you want to use an initial parameter? for practicality. Hardcoding information in the servlet code means that you have to recompile the servlet if the information changes. A web.xml file is plain text. You can edit its content easily using a text editor.

The code that retrieves the initial parameter name and values is given in the following code.

To retrieve initial parameters, you need the ServletConfig object passed by the servlet container to the servlet. After you get the ServletConfig object, you then can use its getInitParameterNames and

`getInitParameter` methods. The `getInitParameterNames` does not take an argument and returns an Enumeration containing all the parameter names in the `ServletConfig` object. The `getInitParameter` takes a String containing the parameter name and returns a String containing the value of the parameter.

Because the servlet container passes a `ServletConfig` object to the `init` method, it is easiest to write the code in the `init` method. The code given below, loops through the Enumeration object called `parameters` that is returned from the `getInitParameterNames` method. For each parameter, it outputs the parameter name and value. The parameter value is retrieved using the `getInitParameter` method.

Retrieving Initial Parameters

```
import javax.servlet.*;
import java.util.Enumeration;
import java.io.IOException;

public class ConfigDemoServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
        Enumeration parameters = config.getInitParameterNames();
        while (parameters.hasMoreElements()) {
            String parameter = (String) parameters.nextElement();
            System.out.println("Parameter name : " + parameter);
            System.out.println("Parameter value : " + config.getInitParameter(parameter));
        }
    }
    public void destroy() {
    }
    public void service(ServletRequest request, ServletResponse response) throws
ServletException, IOException {
    }
    public String getServletInfo() {
        return null;
    }
    public ServletConfig getServletConfig() {
        return null;
    }
}
```

The output of the code in the console is as follows:

```
Parameter name : adminContactNumber
Parameter value : 09005017996
Parameter name : adminEmail
Parameter value : tom@abc.com
```

Preserving the ServletConfig

The code given above, shows how you can use the ServletConfig object in the init method. Sometimes, however, you may want more flexibility. For example, you may want to have access to the ServletConfig object from the service method, when you are servicing the user. In this case, you need to preserve the ServletConfig object to a class level variable. This task is not difficult. You need to create a ServletConfig object variable and set it to the ServletConfig object returned by the servlet container in the init method.

The following code gives the code that preserves the ServletConfig object for later use. First, you need a variable for the ServletConfig object.

```
ServletConfig servletConfig;
```

Then, in the init method, you write the following code:

```
servletConfig = config;
```

Now the servletConfig variable references the ServletConfig object returned by the servlet container. The getServletConfig method is provided to do just that: return the ServletConfig object.

```
public ServletConfig getServletConfig() {  
    return servletConfig;  
}
```

If you extend the ReserveConfigServlet class, you can still retrieve the ServlerConfig object by calling the getServletConfig method.

Preserving the ServletConfig Object

```
import javax.servlet.*;  
import java.io.IOException;  
  
public class ReserveConfigServlet implements Servlet {  
    ServletConfig servletConfig;  
  
    public void init(ServletConfig config) throws ServletException {  
        servletConfig = config;  
    }  
    public void destroy() {}  
    public void service(ServletRequest request, ServletResponse response)  
        throws ServletException, IOException {}  
    public String getServletInfo() {  
        return null; }  
    public ServletConfig getServletConfig() {  
        return servletConfig;  
    } }
```

The Servlet Context

In servlet programming, the servlet context is the environment where the servlet runs. The servlet container creates a `ServletContext` object that you can use to access information about the servlet's environment.

A servlet also can bind an object attribute into the context by name. Any object bound into a context is available to any other servlet that is part of the same web application.

How do you obtain the `ServletContext` object? Indirectly, from the `ServletConfig` object passed by the servlet container to the servlet's `init` method. The `ServletConfig` interface has a method called `getServletContext` that returns the `ServletContext` object. You then can use the `ServletContext` interface's various methods to get the information you need. These methods include the following:

- **getMajorVersion:** This method returns an integer representing the major version for the servlet API that the servlet container supports. If the servlet container supports the servlet API version 2.3, this method will return 2.
- **getMinorVersion:** This method returns an integer representing the minor version of the servlet API that the servlet container supports. For the servlet API version 2.3, this method will return 3.
- **getAttributeNames:** This method returns an enumeration of strings representing the names of the attributes currently stored in the `ServletContext`.
- **getAttribute:** This method accepts a String containing the attribute name and returns the object bound to that name.
- **setAttribute:** This method stores an object in the `ServletContext` and binds the object to the given name. If the name already exists in the `ServletContext`, the old bound object will be replaced by the object passed to this method.
- **removeAttribute:** This method removes from the `ServletContext` the object bound to a name. The `removeAttribute` method accepts one argument: the name of the attribute to be removed.

The following code shows a servlet named `ContextDemoServlet` that retrieves some of the servlet context information, including attribute names and values, minor and major versions of the servlet container, and the server info.

Retrieving Servlet Context Information

```
import javax.servlet.*;
import java.util.Enumeration;
import java.io.IOException;

public class ContextDemoServlet implements Servlet {
    ServletConfig servletConfig;

    public void init(ServletConfig config) throws ServletException {
        servletConfig = config; }
```

```

public void destroy() {
}
public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException {
ServletContext servletContext = servletConfig.getServletContext();
Enumeration attributes = servletContext.getAttributeNames();
while (attributes.hasMoreElements()) {
    String attribute = (String) attributes.nextElement();
    System.out.println("Attribute name : " + attribute);
    System.out.println("Attribute value : " + servletContext.getAttribute(attribute));
}

System.out.println("Major version : " + servletContext.getMajorVersion());
System.out.println("Minor version : " + servletContext.getMinorVersion());
System.out.println("Server info : " + servletContext.getServerInfo()); }

public String getServletInfo() {
    return null;
}
public ServletConfig getServletConfig() {
    return null;
}
}

```

The output of the code is as follows. This output may be different on your computer, depending on the version of Tomcat you are using, the operating system, and so on.

```

Attribute name : javax.servlet.context.tempdirAttribute value : ..\work\localhost\myApp
Attribute name : org.apache.catalina.resources
Attribute value : org.apache.naming.resources.ProxyDirContext@24e2e3
Attribute name : org.apache.catalina.WELCOME_FILES
Attribute value : [Ljava.lang.String;@2bb7e0
Attribute name : org.apache.catalina.jsp_classpath
Attribute value : C:\tomcat4\webapps\myApp\WEB-INF\classes; .
.
Major version : 2Minor version : 3
Server info : Apache Tomcat/4.0-b5

```

Sharing Information Among Servlets

For some applications, you want to make certain types of information available to all the servlets. You can share this information—such as a database connection string or a page count—among the servlets by using attributes in the `ServletContext` object.

The following example uses two servlets:

AttributeSetterServlet and DisplayAttributesServlet.

The AttributeSetterServlet servlet, shown in the first example, binds the name password to a String object containing the word "ding-dong". The servlet does this by first obtaining the ServletContext object from the ServletConfig object passed by the servlet container to the init method. Then the servlet uses the setAttribute method to bind "password" with "ding-dong".

The AttributeSetterServlet

```
import javax.servlet.*;
import java.io.IOException;

public class AttributeSetterServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
        // bind an object that is to be shared among other servlets
        ServletContext servletContext = config.getServletContext();
        servletContext.setAttribute("password", "dingdong");    }

    public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {
    }

    public void destroy() {
    }

    public String getServletInfo() {
        return null;
    }
    public ServletConfig getServletConfig() {
        return null;
    }
}
```

The following code is the servlet that retrieves all attribute name/value pairs in the ServletContext object. The init method of this servlet preserves the ServletConfig object into servletConfig. The service method then uses the ServletConfig interface's getServletContext method to obtain the ServletContext object. After you get the ServletContext object, you can then use its getAttributeNames method to get an Enumeration of all attribute names and loop through it to obtain each attribute's value, which it outputs to the console along with the attribute name.

DisplayAttributesServlet

```
import javax.servlet.*;
import java.io.IOException;
import java.util.Enumeration;

public class ResponseDemoServlet implements Servlet {
    ServletConfig servletConfig;
    public void init(ServletConfig config) throws ServletException {
        servletConfig = config;
    }

    public void destroy() {
    }

    public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {

        ServletContext servletContext = servletConfig.getServletContext();
        Enumeration attributes = servletContext.getAttributeNames();
        while (attributes.hasMoreElements()) {
            String attribute = (String) attributes.nextElement();
            System.out.println("Attribute name : " + attribute);
            System.out.println("Attribute value : " + servletContext.getAttribute(attribute));
        }
    }

    public String getServletInfo() {
        return null;
    }

    public ServletConfig getServletConfig() {
        return null;
    }

    Enumeration attributes = servletContext.getAttributeNames();
    while (attributes.hasMoreElements()) {
        String attribute = (String) attributes.nextElement();
        System.out.println("Attribute name : " + attribute);
        System.out.println("Attribute value : " + servletContext.getAttribute(attribute));
    }
}
```

To see the servlets work, first you need to call the AttributeSetterServlet servlet to set the attribute "password". You then call the DisplayAttributesServlet to get an Enumeration of the names of all attributes and display the values.

The output is given here:

```
Attribute name : javax.servlet.context.tempdir  
Attribute value : C:\123data\JavaProjects\JavaWebBook\work\localhost_8080  
Attribute name : password  
Attribute value : dingdong  
Attribute name : sun.servlet.workdir  
Attribute value : C:\123data\JavaProjects\JavaWebBook\work\localhost_8080
```

Requests and Responses

Requests and responses are what a web application is all about. In a servlet application, a user using a web browser sends a request to the servlet container, and the servlet container passes the request to the servlet.

In a servlet paradigm, the user request is represented by the ServletRequest object passed by the servlet container as the first argument to the service method. The service method's second argument is a ServletResponse object, which represents the response to the user.

The ServletRequest Interface

The ServletRequest interface defines an object used to encapsulate information about the user's request, including parameter name/value pairs, attributes, and an input stream.

The ServletRequest interface provides important methods that enable you to access information about the user. For example, the getParameterNames method returns an Enumeration containing the parameter names for the current request. To get the value of each parameter, the ServletRequest interface provides the getParameter method.

The getRemoteAddress and getRemoteHost methods are two methods that you can use to retrieve the user's computer identity. The first returns a string representing the IP address of the computer the client is using, and the second method returns a string representing the qualified host name of the computer.

The following example, shows a ServletRequest object in action. The example consists of an HTML form in a file named index.html that you need to put in the application directory—that is, under myApp—and a servlet called RequestDemoServlet.

index.html

```
<HTML>  
<HEAD>  
<TITLE>Sending a request</TITLE>  
</HEAD>  
<BODY>
```

```

<FORM ACTION=servlet/RequestDemoServlet METHOD="POST"> <BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>

```

RequestDemoServlet

```

import javax.servlet.*;
import java.util.Enumeration;
import java.io.IOException;

public class RequestDemoServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException { }
    public void destroy() { }

    public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {
        System.out.println("Server Port: " + request.getServerPort());
        System.out.println("Server Name: " + request.getServerName());
        System.out.println("Protocol: " + request.getProtocol());
        System.out.println("Character Encoding: " + request.getCharacterEncoding());
        System.out.println("Content Type: " + request.getContentType());
        System.out.println("Content Length: " + request.getContentLength());
        System.out.println("Remote Address: " + request.getRemoteAddr());
        System.out.println("Remote Host: " + request.getRemoteHost());
        System.out.println("Scheme: " + request.getScheme());
        Enumeration parameters = request.getParameterNames();
        while (parameters.hasMoreElements()) {
            String parameterName = (String) parameters.nextElement();
            System.out.println("Parameter Name: " + parameterName);
            System.out.println("Parameter Value: " + request.getParameter(parameterName));    }
        Enumeration attributes = request.getAttributeNames();
        while (attributes.hasMoreElements()) {
            String attribute = (String) attributes.nextElement();
            System.out.println("Attribute name: " + attribute);
            System.out.println("Attribute value: " + request.getAttribute(attribute));
        }
    }

    public String getServletInfo() {
        return null;    }
}

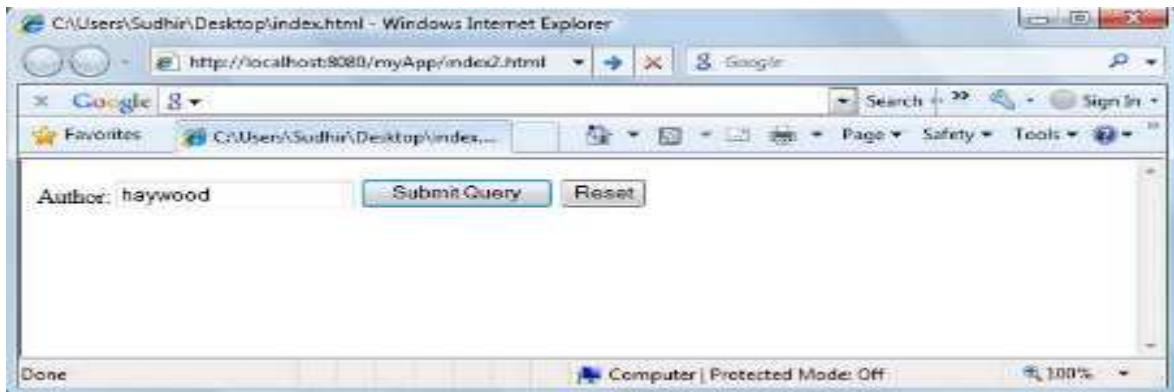
```

```
public ServletConfig getServletConfig() {  
    return null;  
}  
}
```

To run the example, first request the index.html file by using the following URL:

`http://localhost:8080/myApp/index.html`

Following figure shows the index.html file in which "haywood" has been typed in as the value for author.



When you submit the form, you should see the list of attribute names and values in your console.



The ServletResponse Interface

The `ServletResponse` interface represents the response to the user. The most important method of this interface is `getWriter`, from which you can obtain a `java.io.PrintWriter` object that you can use to write HTML tags and other text to the user. The code given below offer an HTML file named `index2.html` and a servlet whose service method is overridden with code that outputs some HTML tags to the user.

This servlet modifies the example given above and instead of sending the information to the console, the service method sends it back to the user.

index2.html

```
<HTML>
<HEAD>
<TITLE>Sending a request</TITLE>
</HEAD>
<BODY>
<FORM ACTION=servlet/ResponseDemoServlet METHOD="POST"> <BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset"> </FORM>
</BODY>
</HTML>
```

The ResponseDemoServlet

```
import javax.servlet.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Enumeration;

public class ResponseDemoServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException { }
    public void destroy() {
    }
    public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>");
        out.println("ServletResponse");
        out.println("</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<B>Demonstrating the ServletResponse object</B>");
        out.println("<BR>");
        out.println("<BR>Server Port: " + request.getServerPort());
        out.println("<BR>Server Name: " + request.getServerName());
        out.println("<BR>Protocol: " + request.getProtocol());
    }
}
```

```

out.println("<BR>Character Encoding:" + request.getCharacterEncoding());
out.println("<BR>Content Type: " + request.getContentType());
out.println("<BR>Content Length: " + request.getContentLength());
out.println("<BR>Remote Address: " + request.getRemoteAddr());
out.println("<BR>Remote Host: " + request.getRemoteHost());
out.println("<BR>Scheme: " + request.getScheme());
Enumeration parameters = request.getParameterNames();
while (parameters.hasMoreElements()) {
    String parameterName = (String) parameters.nextElement();
    out.println("<br>Parameter Name: " + parameterName);
    out.println("<br>Parameter Value: " +
               request.getParameter(parameterName));    }
Enumeration attributes = request.getAttributeNames();
while (attributes.hasMoreElements()) {
    String attribute = (String) attributes.nextElement();
    out.println("<BR>Attribute name: " + attribute);
    out.println("<BR>Attribute value: " + request.getAttribute(attribute));    }
out.println("</BODY>");
out.println("</HTML>");
}
public String getServletInfo() {
    return null;
}
public ServletConfig getServletConfig() {
    return null;
}
}

```

To run the example, first request the index2.html file by using the following URL:

http://localhost:8080/myApp/index2.html

The GenericServlet Wrapper Class

By now, you have been creating servlet classes that implement the javax.servlet.Servlet interface. Everything works fine, but there are two annoying things that you've probably noticed:

1. You have to provide implementations for all five methods of the Servlet interface, even though most of the time you only need one. This makes your code look unnecessarily complicated.
2. The ServletConfig object is passed to the init method. You need to preserve this object to use it from other methods. This is not difficult, but it means extra work.

The javax.servlet package provides a wrapper class called GenericServlet that implements two important interfaces from the javax.servlet package: Servlet and ServletConfig, as well as the java.io.Serializable interface. The GenericServlet class provides implementations for all methods, most of which are blank. You can extend GenericServlet and override only methods that you need to use. Clearly, this looks like a better solution.

The code given below is a servlet called SimpleServlet that extends GenericServlet. The code provides the implementation of the service method that sends some output to the browser. Because the service method is the only method you need, only this method needs to appear in the class. Compared to all servlet classes that implement the javax.servlet.Servlet interface directly, SimpleServlet looks much cleaner and clearer.

Extending GenericServlet

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;

public class SimpleServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("Extending GenericServlet");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Extending GenericServlet makes your code simpler.");
    out.println("</BODY>");
    out.println("</HTML>");
}}
```

The output from the SimpleServlet servlet is shown below.



Creating Thread-Safe Servlets

A servlet container allows multiple requests for the same servlet by creating a different thread to service each request. In many cases, each thread deals with its own `ServletRequest` and `ServletResponse` objects that are isolated from other threads. Problems start to arise, however, when your servlet needs to access an external resource. To understand the problem introduced by multithreaded servlets, consider the following "playing dog" illustration.

Imagine a servlet accessing an external resource as a dog who enjoys moving tennis balls from one box to another. Each box can hold ten balls, no matter how the balls are arranged. The boxes and the balls are an external resource to the dog. To play, the dog needs two boxes and ten balls. Initially, those ten balls are placed in the first box. The dog moves all balls from the first box to the second, one ball at a time. The dog is smart enough to count to ten. Therefore, it knows when it's finished.

Now imagine a second thread of the same servlet as a second dog that plays the same game. Because there are only two boxes and ten balls for both dogs, the two dogs share the same "external resource." The game goes like this:

1. The first dog starts first (the servlet receives a call from a user).
2. After the first dog moves three balls, the second dog starts to play (the servlet is invoked by the second user). What will happen?

The two dogs sharing the same balls are illustrated in the following figure.



The first dog and the second dog will not find enough balls to finish the game, and both will be confused.

If somehow the second dog can be queued to wait to start until the first dog finishes, however, the two dogs are happy.

That's what happens when two threads of the same servlet need to access an external resource, such as opening a file and writing to it. Consider the following example, which reflects a real-world situation.

The code given below presents a page counter servlet. What it does is simple. The servlet overrides the `service` method to do the following:

1. Open the counter.txt file using a BufferedReader, read the number into a counter, and close the file.
2. Increment the counter.
3. Write the counter back to the counter.txt file.
4. Display the counter in the web browser.

Imagine what happens if there are two users, Boni and Bulbul, who request the servlet. First Boni requests it, and then a couple of nanoseconds after Boni, Bulbul requests the same servlet. The scenario probably looks like this:

1. The service method executes Steps 1 and 2, and then gets distracted by the other request.
 2. The method then does Step 1 from Bulbul before continuing to do Step 3 and 4 for Boni.
- What happens next? Boni and Bulbul get the same number, which is not how it is supposed to be. The servlet has produced an incorrect result. As you can see in the following code, servlet is an unsafe multithreaded servlet.

Unsafe Multi-Threaded Servlet

```
import javax.servlet.*;
import java.io.*;

public class SingleThreadedServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        int counter = 0;
        // get saved value
        try {
            BufferedReader reader = new BufferedReader(new FileReader("counter.txt"));
            counter = Integer.parseInt( reader.readLine() );
            reader.close();
        }
        catch (Exception e) { }
        counter++;      // increment counter
        // save new value
        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter("counter.txt"));
            writer.write(Integer.toString(counter));
            writer.close();
        }
        catch (Exception e) { }

        try {
            PrintWriter out = response.getWriter();
            out.println("You are visitor number " + counter);
        }
        catch (Exception e) { } } }
```

To solve the problem, remember the solution to the "playing dog" illustration: When the second dog waited until the first dog finished playing, both dogs could complete the game successfully. This is exactly how you solve the problem in a servlet needing to service two users at the same time: by making the second user wait until the first servlet finishes serving the first user. This solution makes the servlet single-threaded.

This solution is very easy to do because of the marker SingleThreadedServlet interface. You don't need to change your code; you need only to implement the interface.

The following code is the modification of the previous code. Nothing changes, except that the SingleThreadedServlet class now implements SingleThreadModel, making it thread safe.

Safe Multi-Threaded Servlet

```
import javax.servlet.*;
import java.io.*;

public class SingleThreadedServlet extends GenericServlet implements SingleThreadModel {
    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        int counter = 0;
        // get saved value
        try {
            BufferedReader reader = new BufferedReader(new FileReader("counter.txt"));
            counter = Integer.parseInt( reader.readLine() );
            reader.close();
        }
        catch (Exception e) { }
        counter++; // increment counter
        // save new value
        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter("counter.txt"));
            writer.write(Integer.toString(counter));
            writer.close();
        }
        catch (Exception e) { }
        try {
            PrintWriter out = response.getWriter();
            out.println("You are visitor number " + counter);
        }
        catch (Exception e) { } } }
```

Now, if a user requests the service of the servlet while the servlet is servicing another user, the user who comes later will have to wait.

If you want to experience erroneous multi-threading yourself, the code given below provides the SingleThreadedServlet with a delay of 6 seconds. Open two browsers and request the same servlet quickly. Notice that you get the same number for both browsers.

Demonstrating an Unsafe Multi-Threaded Servlet

```
import javax.servlet.*;
import java.io.*;

public class SingleThreadedServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        int counter = 0;
        // get saved value
        try {
            BufferedReader reader = new BufferedReader(new FileReader("counter.txt"));
            counter = Integer.parseInt( reader.readLine() );
            reader.close();
        }
        catch (Exception e) {    }
        counter++;    // increment counter
        // delay for 6 seconds to make observation possible
        try {
            Thread thread = new Thread();
            thread.sleep(6000);
        }
        catch (InterruptedException e) {
        }
        // saved new value
        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter("counter.txt"));
            writer.write(Integer.toString(counter));
            writer.close();
        }
        catch (Exception e) {    }
        try {
            PrintWriter out = response.getWriter();
            out.println("You are visitor number " + counter);    }
        catch (Exception e) {
        }
    }
}
```

What the code does is simple: It opens the counter.txt file, reads the value, increments the value, and writes the incremented value back to the file.

However, between the line of code that increments the value and the line of code that writes the incremented value back to the user, we inserted the following code:

```
try {  
    Thread thread = new Thread();  
    thread.sleep(6000);  
}  
catch (InterruptedException e) {  
}
```

Now you have time to request the same servlet from the second browser. The value shown in both browsers will be the same if the second request comes before the first thread of the servlet has the time to update the value in the counter.txt file.

An inexperienced programmer would wonder whether a good solution might be to make every servlet implement the SingleThreadModel interface. The answer is no. If a servlet never accesses an external resource, queuing the second request will create unnecessary delay to the subsequent user after the first. Also, if the external resource is accessed but there is no need to update its value, you don't need to implement the SingleThreadModel interface. For example, if the service method of a servlet needs only to read a static value from a file, you can let multiple threads of the servlet open and read the file at the same time.

Advanced Servlet Part-I

In the previous chapters, you have learned how to write servlets, run them in the servlet container, and invoke them from a web browser. You also have studied various classes and interfaces in the javax.servlet package and learned how to solve the problem introduced by a multi-threaded servlet.

When you are programming servlets, however, you will work with another package called javax.servlet.http. The classes and interfaces in this package derive from those in javax.servlet; however, the members of the javax.servlet.http package are much richer and more convenient to use. In this package, the HttpServlet class represents a servlet, extending javax.servlet.GenericServlet and bringing a great number of methods of its own. The javax.servlet.http package also has interfaces that are equivalent to javax.servlet.ServletRequest and javax.servlet.ServletResponse interfaces—the HttpServletRequest and the HttpServletResponse, respectively. It is not a coincidence that HttpServletRequest extends the javax.servlet.ServletRequest interface, and HttpServletResponse is derived from the javax.servlet.ServletResponse interface.

Additional classes exist that are not available in the javax.servlet package. For example, you can use a class called Cookie to work with cookies. In addition, you will find session-related methods in the HttpServlet class that enable you to deal with user sessions.

Let's now begin with an overview of the HttpServlet class, a class that you almost always extend when developing servlets.

The HttpServlet Class

As mentioned previously, the HttpServlet class extends the javax.servlet.GenericServlet class. The HttpServlet class also adds a number of interesting methods for you to use. The most important are the six doxxx methods that get called when a related HTTP request method is used. The six methods are doPost, doGet, doPut, doDelete, doOptions and doTrace. Each doxxx method is invoked when a corresponding HTTP method is used. For instance, the doGet method is invoked when the servlet receives an HTTP request that was sent using the GET method.

Note: If you are familiar with the HTTP 1.1 protocol, you will notice that the HEAD method does not have a corresponding do method in the servlet. You are right. Actually, there is a doHead method in the HttpServlet class, but it is a private method.

Of the six doxxx methods, the doPost and the doGet methods are the most frequently used.

The doPost method is called when the browser sends an HTTP request using the POST method. The POST method is one of the two methods that can be used by an HTML form. Consider the following HTML form at the client side:

```
<FORM ACTION="Register" METHOD="POST">
<INPUT TYPE=TEXT Name="firstName">
<INPUT TYPE=TEXT Name="lastName">
<INPUT TYPE=SUBMIT>
</FORM>
```

When the user clicks the Submit button to submit the form, the browser sends an HTTP request to the server using the POST method. The web server then passes this request to the registered servlet and the doPost method of the servlet is invoked. Using the POST method in a form, the parameter name/value pairs of the form are sent in the request body. For example, if you use the preceding form as an example and enter Ann as the value for firstName and Go as the value for lastName, you will get the following result in the request body:

```
firstName=Ann
lastName=Go
```

An HTML form can also use the GET method; however, POST is much more often used with HTML forms.

The doGet method is invoked when an HTTP request is sent using the GET method. GET is the default method in HTTP. When you type a URL, such as www.yahoo.com, your request is sent to Yahoo! using the GET method. If you use the GET method in a form, the parameter name/value pairs are appended to the URL. Therefore, if you have two parameters named firstName and lastName in your form, and the user enters Ann and Go, respectively, the URL to your servlet will become something like the following:

```
http://yourdomain/myApp/Register?firstName=Ann&lastName=Go
```

Upon receiving a GET method, the servlet will call its doGet method.

Note: You may wonder how a servlet knows what doxxx method to invoke. You can find the answer by reading the source code of the HttpServlet class. This class inherits the service method from the javax.servlet.Servlet interface that gets called by the servlet container. Remember that its signature is as follows:

```
public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException
```

The method tries to downcast request to HttpServletRequest and response to HttpServletResponse, and pass both as arguments to the second service method that has the following signature:

```
protected void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
```

The HttpServletRequest interface has a method named getMethod that returns a String containing the HTTP method used by the client request. Knowing the HTTP method, the service method simply calls the corresponding doxxx method. The following servlet code demonstrates the doGet and the doPost methods.

Note: If an HTML form does not have the ACTION attribute, the default value for this attribute is the current page.

The doGet and doPost Methods

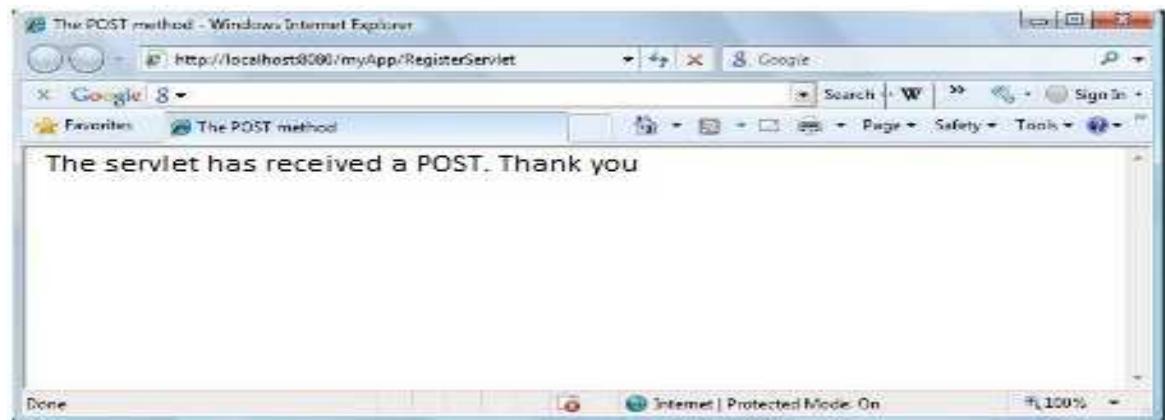
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class RegisterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>The GET method</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("The servlet has received a GET. " + "Now, click the button below.");
        out.println("<BR>");
        out.println("<FORM METHOD=POST>");
        out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
        out.println("</FORM>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
    public void doPost(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>The POST method</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("The servlet has received a POST. Thank you.");
        out.println("</BODY>");
```

```
    out.println("</HTML>");  
}  
}  
}
```

When the servlet is first called from a web browser by typing the URL to the servlet in the Address or Location box, GET is used as the request method. At the server side, the doGet method is invoked. The servlet sends a string saying "The servlet has received a GET. Now, click the button below." plus an HTML form. The output is shown below.



The form sent to the browser uses the POST method. When the user clicks the button to submit the form, a POST request is sent to the server. The servlet then invokes the doPost method, sending a String saying, "The servlet has received a POST. Thank you," to the browser. The output of doPost is shown below.



The HttpServletRequest Interface

In addition to providing several more protocol-specific methods in the HttpServlet class, the javax.servlet.http package also provides more sophisticated request and response interfaces. The request interface, HttpServletRequest, is described in this section. The response interface, HttpServletResponse, is explained in the next section.

Obtaining HTTP Request Headers from HttpServletRequest

The HTTP request that a client browser sends to the server includes an HTTP request header with important information, such as cookies and the referer. You can access these headers from the HttpServletRequest object passed to a doxxx method.

The following example demonstrates how you can use the HttpServletRequest interface to obtain all the header names and sends the header name/value pairs to the browser.

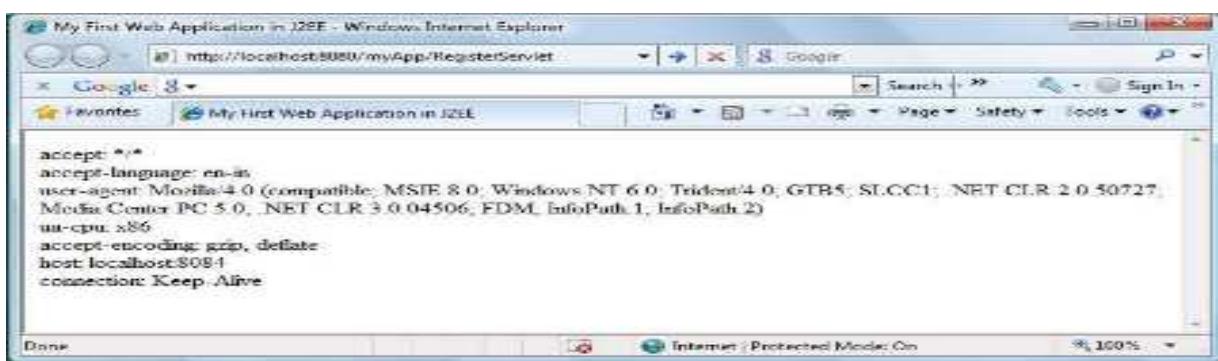
Obtaining HTTP request Headers

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class RegisterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration enumeration = request.getHeaderNames();
        while (enumeration.hasMoreElements()) {
            String header = (String) enumeration.nextElement();
            out.println(header + ": " + request.getHeader(header) + "<BR>");      }
    }
}
```

The RegisterServlet uses the getHeaderNames and the getHeader methods. The getHeaderNames is first called to obtain an Enumeration containing all the header names found in the client request. The value of each header then is retrieved by using the getHeader method, passing a header name.

The output of the above code depends on the client environment, such as the browser used and the operating system of the client's machine. For example, some browsers might send cookies to the server. Also, whether the servlet is requested by the user typing the URL in the Address/Location box or by clicking a hyperlink also accounts for the presence of an HTTP request header called referer.



Some other methods of the `HttpServletRequest` interface provide information about paths. The `getPathInfo` method returns—following the servlet path but preceding the query string—a String containing any additional path information, or returns null if there is no additional path information. The `getPathTranslated` method returns the same information as the `getPathInfo` method, but translates the path to its physical path name before returning it, or returns null if there is no additional path information. Additional information comes after the servlet name and before the query string. The servlet name and the additional information is separated by the forward slash character (/).

For example, consider a request to a servlet called `PathInfoDemoServlet`, made with the following URL:

```
http://localhost:8080/myApp/servlet/PathInfoDemoServlet/AddInfo?id=2
```

This URL contains additional info "/AddInfo" after the servlet name. The `getPathInfo` method will return the String "/AddInfo", the `getqueryString` method will return "id=2", and the `getPathTranslated` method returns "C:\App\Java\AddInfo". The return value of `getPathTranslated` depends on the location of the servlet class file.

Next, the `getRequestURI` method returns the first line of the request's Uniform Resource Identifier (URI). This is the part of the URI that is found to the left of the query string. The `getServletPath` method returns the part of the URI that refers to the servlet being invoked.

Obtaining the Query String from `HttpServletRequest`

The next important method is the `getqueryString` method, which is used to retrieve the query string of the HTTP request. A query string is the string on the URL to the right of the path to the servlet. The following example helps you see what a query string looks like. As mentioned previously, if you use the GET method in an HTML form, the parameter name/value pairs will be appended to the URL. The following code is a servlet named `HttpRequestDemoServlet` that displays the value of the request's query string and a form.

Obtaining the Query String

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

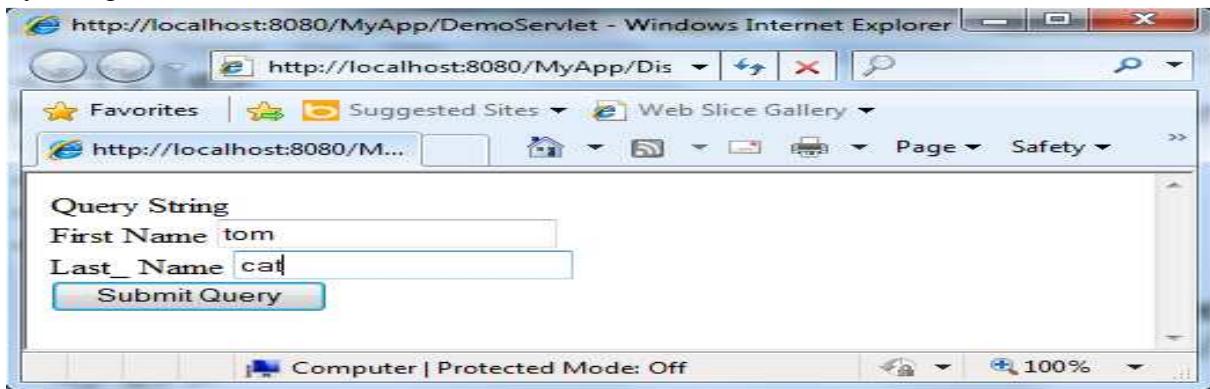
public class HttpRequestDemoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
```

```

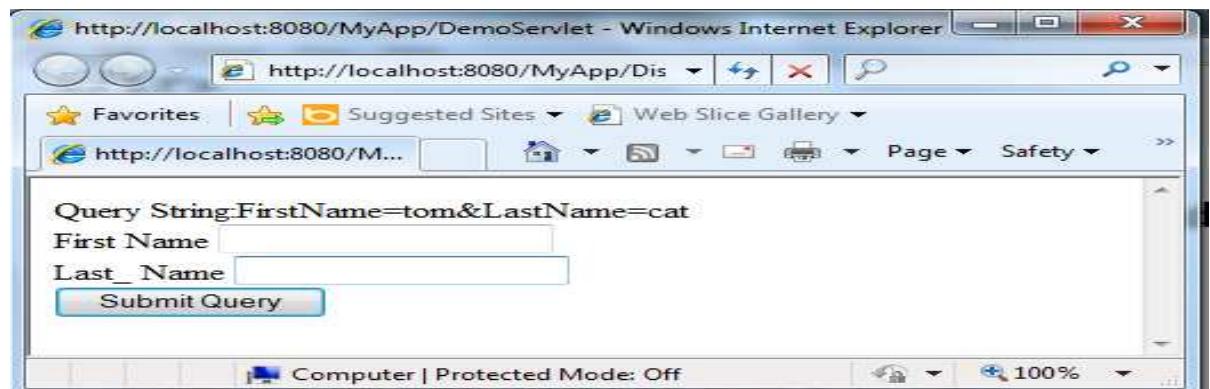
out.println("<TITLE>Obtaining the Query String</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("Query String: " + request.getQueryString() + "<BR>");
out.println("<FORM METHOD=GET>");
out.println("<BR>First Name: <INPUT TYPE=TEXT NAME=FirstName>");
out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
out.println("</FORM>");
out.println("</BODY>");
out.println("</HTML>");
}
}

```

When the user enters the URL to the servlet in the web browser and the servlet is first called, the query string is null, as shown below.



After you enter some values into the HTML form and submit the form, the page is redisplayed. Note that now there is a string added to the URL. The query string has a value of the parameter name/value pairs separated by an ampersand (&). The page is shown below.



Obtaining the Parameters from HttpServletRequest

You have seen that you can get the query string containing a value. This means that you can get the form parameter name/value pairs or other values from the previous page. You should not use the

`getQueryString` method to obtain a form's parameter name/value pairs, however, because this means you have to parse the string yourself. You can use some other methods in `HttpServletRequest` to get the parameter names and values: the `getParameterNames` and the `getParameter` methods.

The `getParameterNames` method returns an `Enumeration` containing the parameter names. In many cases, however, you already know the parameter names, so you don't need to use this method. To get a parameter value, you use the `getParameter` method, passing the parameter name as the argument.

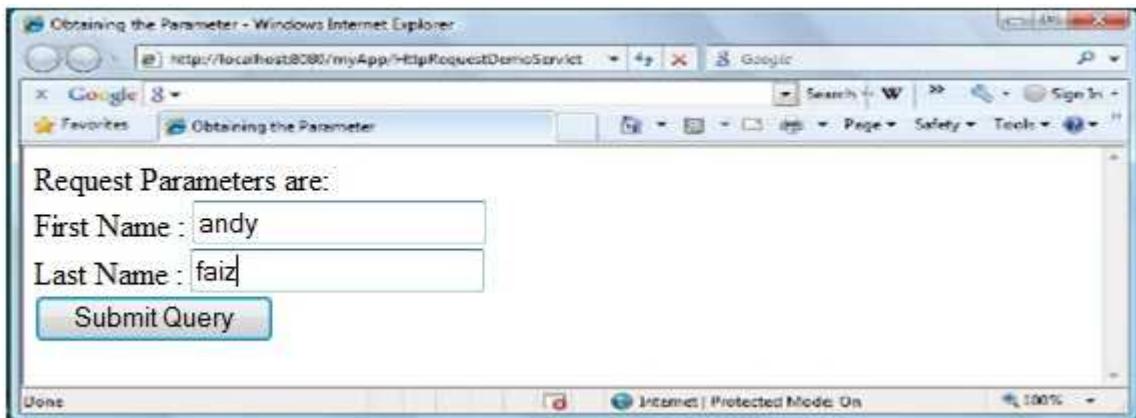
The following example demonstrates how you can use the `getParameterNames` and the `getParameter` methods to display all the parameter names and values from the HTML form from the previous page. The code is given below.

Obtaining the Parameter Name/Value Pairs

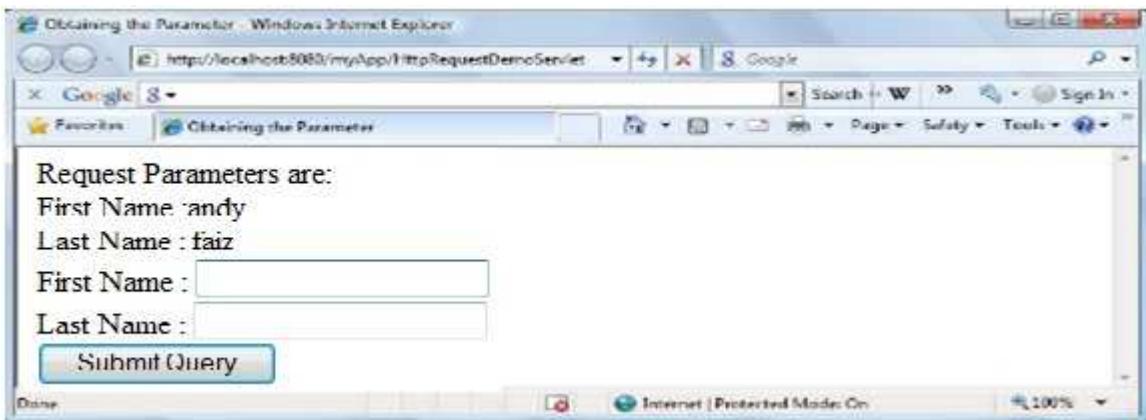
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class HttpRequestDemoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Obtaining the Parameter</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("Request Parameters are:<BR>");
        Enumeration enumeration = request.getParameterNames();
        while (enumeration.hasMoreElements()){
            String parameterName = (String) enumeration.nextElement();
            out.println(parameterName + " : " + request.getParameter(parameterName) + "<BR> ");
        }
        out.println("<FORM METHOD=GET>");
        out.println("<BR>First Name: <INPUT TYPE=TEXT NAME=FirstName>");
        out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
        out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
        out.println("</FORM>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

When the servlet is first called, it does not have any parameter from the previous request. Therefore, the no parameter name/value pair is displayed, as shown below.



On subsequent requests, the user should enter values for both the firstName and lastName parameters. This is reflected on the next page, which is shown below.



The servlet code given above can also be used without any modification if the form uses the POST method, which is what you normally use for a form. There are numerous cases, however, where you need to pass non-form values in the URL. This technique is discussed further.

Manipulating Multi-Value Parameters

You may have a need to use parameters with the same name in your form. This case might arise, for example, when you are using check box controls that can accept multiple values or when you have a multiple-selection HTML select control. In situations like these, you can't use the getParameter method because it will give you only the first value. Instead, you use the getParameterValues method.

The getParameterValues method accepts one argument: the parameter name. It returns an array of string containing all the values for that parameter. If the parameter of that name is not found, the getParameterValues method will return a null.

The following example illustrates the use of the getParameterValues method to get all favorite music selected by the user. The code for this servlet is given below.

Obtaining Multiple Values from a Parameter

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class HttpRequestDemoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Obtaining Multi-Value Parameters</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");

        out.println("<BR>");
        out.println("<BR>Select your favorite music:");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<BR><INPUT TYPE=CHECKBOX " + "NAME=favoriteMusic
                  VALUE=Rock>Rock");
        out.println("<BR><INPUT TYPE=CHECKBOX " + "NAME=favoriteMusic
                  VALUE=Jazz>Jazz");
        out.println("<BR><INPUT TYPE=CHECKBOX " + "NAME=favoriteMusic
                  VALUE=Classical>Classical");
        out.println("<BR><INPUT TYPE=CHECKBOX " + "NAME=favoriteMusic
                  VALUE=Country>Country");
        out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
        out.println("</FORM>");
        out.println("</BODY>");
        out.println("</HTML>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String[] values = request.getParameterValues("favoriteMusic");
        response.setContentType("text/html");
    }
}
```

```

PrintWriter out = response.getWriter();

if(values != null ) {
    int length = values.length;
    out.println("You have selected: ");
    for (int i=0; i<length; i++) {
        out.println("<BR>" + values[i]);
    }
}
}
}
}

```

When the servlet is first called, the doGet method is invoked and the method sends a form to the web browser. The form has four check box controls with the same name: favoriteMusic. Their values are different, however. This is shown below.



When the user selects the value(s) of the check boxes, the browser sends all selected values. On the server side, you use the getParameterValues to retrieve all values sent in the request.

HttpServletResponse

The HttpServletResponse interface provides several protocol-specific methods not available in the javax.servlet.ServletResponse interface.

The HttpServletResponse interface extends the javax.servlet.ServletResponse interface. In the examples in this chapter so far, you have seen that you always use two of the methods in HttpServletResponse when sending output to the browser: setContentType and getWriter.

```

response.setContentType("text/html");
PrintWriter out = response.getWriter();

```

There is more to it, however. The addCookie method sends cookies to the browser. You also use methods to manipulate the URLs sent to the browser. These methods are explored further in the section on user session management.

Another interesting method in the `HttpServletResponse` interface is the `setHeader` method. This method allows you to add a name/value field to the response header.

You can also use a method to redirect the user to another page: `sendRedirect`. When you call this method, the web server sends a special message to the browser to request another page. Therefore, there is always a round trip to the client side before the other page is fetched. This method is used frequently and its use is illustrated in the following example.

Following example shows a Login page that prompts the user to enter a user name and a password. If both are correct, the user will be redirected to a Welcome page. If not, the user will see the same Login page.

When the servlet is first requested, the servlet's `doGet` method is called. The `doGet` method then outputs the form. The user can then enter the user name and password, and submit the form. Note that the form uses the POST method, which means that at the server side, the `doPost` method is invoked and the user name and password are checked against some predefined values. If the user name and password match, the user is redirected to a Welcome page. If not, the `doPost` method outputs the Login form again along with an error message.

A Login Page

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class LoginServlet extends HttpServlet {
    private void sendLoginForm(HttpServletRequest response,      boolean withErrorMessage)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Login</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");

        if (withErrorMessage)
            out.println("Login failed. Please try again.<BR>");
        out.println("<BR>");
        out.println("<BR>Please enter your user name and password.");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<BR>User Name: <INPUT TYPE=TEXT NAME=userName>");
        out.println("<BR>Password: <INPUT TYPE=PASSWORD NAME=password>");
        out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
```

```

out.println("</FORM>");
out.println("</BODY>");
out.println("</HTML>"); }
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    sendLoginForm(response, false);
}

public void doPost(HttpServletRequest request,    HttpServletResponse response)
    throws ServletException, IOException {
    String userName = request.getParameter("userName");
    String password = request.getParameter("password");
    if(userName!=null && password!=null && userName.equals("jamesb") &&
    password.equals("007")) {
        response.sendRedirect("http://domain/app/WelcomePage");
    } else {
        sendLoginForm(response, true);
    }
}
}

```

Note: If you are redirecting to a resource in the same application, you don't need to specify the complete URL; that is, you can just write, in the previous example, `response.sendRedirect ("app/WelcomePage")`. For efficiency, however, you don't normally use the `sendRedirect` method to redirect a user to another resource in the same application. Instead, you forward the user, as you will see in the section, "Request Dispatching," later in this chapter.

In the above servlet code, A private method called `sendLoginForm` has been written that accepts an `HttpServletResponse` object and a boolean that signals whether an error message be sent along with the form. This `sendLoginForm` method is called both from the `doGet` and the `doPost` methods. When called from the `doGet` method, no error message is given, because this is the first time the user requests the page. The `withErrorMessage` flag is therefore false. When called from the `doPost` method, this flag is set to true because the `sendLoginForm` method is only invoked from `doPost` if the user name and password did not match.

The Login page when first requested.



The Login page after a failed login.



After seeing the example, an experienced reader may ask, "If we can go to the Welcome page by just typing its URL in the web browser, why do we have to log in?"

This is true. The user can bypass the Login page, and this issue has to do with session management and has been addressed further.

Sending an Error Code

The `HttpServletResponse` also allows you to send pre-defined error messages. The interface defines a number of public static final integers that all start with `SC_`. For example, `SC_FORBIDDEN` will be translated into an HTTP error 403.

Along with the error code, you also can send a custom error message. Instead of redisplaying the Login page when a failed login occurs, you can send an HTTP error 403 plus your error message. To do this, replace the call to the `sendLoginForm` in the `doPost` method with the following:

```
response.sendError(response.SC_FORBIDDEN, "Login failed.");
```

The user will see the following screen when a login fails.



Sending Special Characters

Several characters have a special meaning in HTML. For instance, the less-than character (<) is used as the opening character of an HTML tag, and the greater-than character (>) is the closing character of an HTML tag.

When sending these characters to be displayed in the browser, you need to encode them so that they will be rendered correctly. For example, consider the following code.

The doGet method of the SpecialCharacterServlet is very simple. It is intended to send a string that will be rendered as the following text in the browser:

In HTML, you use
 to change line.

Incorrect Rendering of Special Characters

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class SpecialCharacterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>HTML Tutorial — Changing Line</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("In HTML, you use <BR> to change line.");
        out.println("</BODY>");
```

```
    out.println("</HTML>");  
}  
}  
}
```

This code produces a problem, however. To see what went wrong, take a look at the output of the doGet method in a browser, as shown below.



Because
 means change line in HTML, the intended string is not displayed correctly. Instead, it was interpreted as a command to cut the original string into two and the output was displayed in two lines.

To get around this, every time you want to display a special character, you need to encode it. The less-than character (<) is encoded as "<" and the greater-than character (>) as ">". Other special characters are the ampersand (&) and double quotation mark ("") characters. You replace the ampersand (&) with "&" and the double quotation marks ("") with """. Additionally, two or more white spaces are always displayed as a single space, unless you convert each individual space to " ".

Converting every occurrence of a special character is a tedious task, however. That's why you need a function that will do it automatically. Such a function is called encodeHtmlTag and is given in following servlet code. Now, if you suspect that the String you want to send to the browser contains a special character, just pass it to the encodeHtmlTag function.

The encodeHtmlTag Function

```
public static String encodeHtmlTag(String tag) {  
    if (tag==null)  
        return null;  
    int length = tag.length();  
    StringBuffer encodedTag = new StringBuffer(2 * length);  
    for (int i=0; i<length; i++) {  
        char c = tag.charAt(i);  
        if (c=='<')  
            encodedTag.append("<");  
        else if (c=='>')  
            encodedTag.append(">");  
        else if (c=='&')  
            encodedTag.append("&amp;");  
    }  
    return encodedTag.toString();  
}
```

```

        else if(c=='''')
            encodedTag.append(""");
        else if(c==' ')
            encodedTag.append(" ");
        else
            encodedTag.append(c);
    }
    return encodedTag.toString();
}

```

Following servlet code demonstrates a servlet that includes the encodeHtmlTag method and uses it to encode any String with special characters.

Using the encodeHtmlTag Method in a Servlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class SpecialCharacterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>HTML Tutorial — Changing Line</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println(encodeHtmlTag("In HTML, you use <BR> to change line."));
        out.println("</BODY>");
        out.println("</HTML>");
    }

    /**
     * Encode an HTML tag so it will be displayed as it is on the browser. Particularly, this method
     * searches the passed in String and replace every occurrence of the following character:
     * '<' with "&lt;"
     * '>' with "&gt;"
     * '&' with "&amp;"
     * '""' with "&quot;"
     */

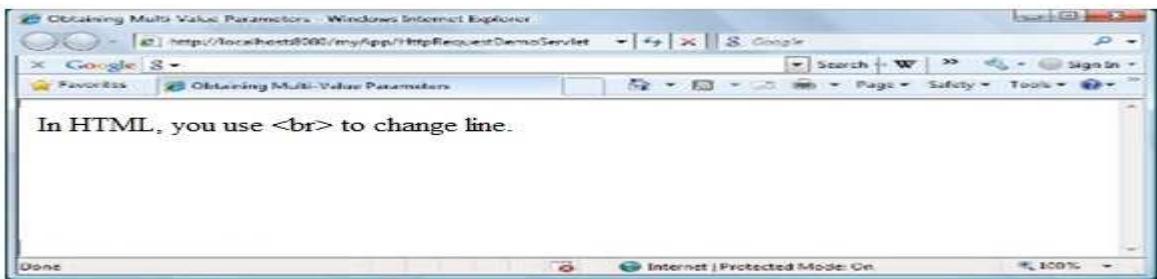
```

```

* '' with " "*
*/
public static String encodeHtmlTag(String tag) {
if(tag==null)
    return null;
int length = tag.length();
StringBuffer encodedTag = new StringBuffer(2 * length);
for (int i=0; i<length; i++) {
    char c = tag.charAt(i);
    if(c=='<')
        encodedTag.append("&lt;");
    else if(c=='>')
        encodedTag.append("&gt;");
    else if(c=='&')
        encodedTag.append("&amp;");
    else if(c=='"')
        encodedTag.append("&quot;"); //when trying to output text as tag's value as in
//values="???".
    else if(c==' ')
        encodedTag.append("&nbsp;");
    else
        encodedTag.append(c);
}
return encodedTag.toString();
}
}

```

Following figure shows the output of sending the string, "In HTML, you use
 to change line". If you look at the HTML source code, you will notice that the < character has been converted to < and the > character to >.



Buffering the Response

If response buffering is enabled, the output to the browser is not sent until the servlet processing is finished or the buffer is full. Buffering enhances the performance of your servlet because the servlet needs to send the string output only once, instead of sending it every time the print or println

method of the PrintWriter object is called. By default, buffering is enabled and the buffer size is 8,192 characters. You can change this value by using the HttpServletResponse interface's setBufferSize method. This method can be called only before any output is sent.

Populating HTML Elements

One of the tasks that you will perform often is populating the values of HTML elements. This is a straightforward task that can be tricky if you are not cautious. To do it correctly, pay attention to the following two rules:

1. Always enclose a value with double quotation marks (""). This way, white spaces will be rendered correctly.
2. If the value contains a double quotation mark character, you need to encode the double quotation marks ("").

The following servlet code contains an HTML form with two elements: a Textbox and a Password box. The Textbox element is given the value, Duncan "The Great" Young, and the password is lo&&lita.

Populating HTML Elements

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class PopulateValueServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String userName = "Duncan \"The Great\" Young";
        String password = "lo&&lita";
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Populate HTML elements</TITLE>");
        out.println("</HEAD>");
        out.println("<H3>Your user name and password.</H3>");
        out.println("<FORM METHOD=POST>");
        out.println("<BR>User name: <INPUT TYPE=TEXT NAME=userName VALUE=\"" +
                  userName + "\">");
        out.println("<BR>Password: <INPUT TYPE=PASSWORD NAME=password VALUE=\"" +
                  password + "\">");
        out.println("<BODY>");
```

```
    out.println("</BODY>");
    out.println("</HTML>");
}
}
```

Request Dispatching

In some circumstances, you may want to include the content from an HTML page or the output from another servlet. Additionally, there are cases that require that you pass the processing of an HTTP request from your servlet to another servlet. The current servlet specification responds to these needs with an interface called RequestDispatcher, which is found in the javax.servlet package. This interface has two methods, which allow you to delegate the request-response processing to another resource: include and forward. Both methods accept a javax.servlet.ServletRequest object and a javax.servlet.ServletResponse object as arguments.

As the name implies, the include method is used to include content from another resource, such as another servlet, a JSP page, or an HTML page. The method has the following signature:

```
public void include(javax.servlet.ServletRequest request, javax.servlet.ServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException
```

The forward method is used to forward a request from one servlet to another. The original servlet can perform some initial tasks on the ServletRequest object before forwarding it. The signature of the forward method is as follows:

```
public void forward(javax.servlet.ServletRequest request, javax.servlet.ServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException
```

The Difference Between sendRedirect and forward

Both the sendRedirect and forward methods bring the user to a new resource. There is a fundamental difference between the two, however, and understanding this can help you write a more efficient servlet.

The sendRedirect method works by sending a status code that tells the browser to request another URL. This means that there is always a round trip to the client side. Additionally, the previous HttpServletRequest object is lost. To pass information between the original servlet and the next request, you normally pass the information as a query string appended to the destination URL.

The forward method, on the other hand, redirects the request without the help from the client's browser. Both the HttpServletRequest object and the HttpServletResponse object also are passed to the new resource.

To perform a servlet include or forward, you first need to obtain a RequestDispatcher object. You can obtain a RequestDispatcher object three different ways, as follows:

- Use the getRequestDispatcher method of the javax.servlet.ServletContext interface, passing a

String containing the path to the other resource. The path is relative to the root of the ServletContext.

- Use the getRequestDispatcher method of the javax.servlet.ServletRequest interface, passing a String containing the path to the other resource. The path is relative to the current HTTP request.
- Use the getNamedDispatcher method of the javax.servlet.ServletContext interface, passing a String containing the name of the other resource.

When programmers new to servlet programming are writing code for request dispatching, they often make the common mistake of passing an incorrect path to the getRequestDispatcher method.

A big difference exists between the getRequestDispatcher method of the ServletContext interface and that belonging to the ServletRequest interface. The one you use depends on the location of the resource to be included or forwarded to. If you use the getRequestDispatcher method of the javax.servlet.ServletContext interface, you pass a path that is relative to the root of the ServletContext. If you use the getRequestDispatcher method of the javax.servlet.ServletRequest interface, you pass a path that is relative to the current HTTP request.

When you are creating a RequestDispatcher object from a servlet named FirstServlet to include or forward the request to another servlet called SecondServlet, the easiest way is to place the class files of both FirstServlet and SecondServlet in the same directory. This way, FirstServlet can be invoked from the URL `http://domain/VirtualDir/servlet/FirstServlet` and SecondServlet can be called from the URL `http://domain/VirtualDir/servlet/SecondServlet`.

You then can use the getRequestDispatcher from the ServletRequest interface, passing the name of the second servlet. In FirstServlet, you can write the following code:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = request.getRequestDispatcher("SecondServlet");
    rd.include(request, response);
}
Or
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = request.getRequestDispatcher("SecondServlet");
    rd.forward(request, response);
}
```

Because both FirstServlet and SecondServlet are in the same directory, you don't need to include the forward slash (/) character before SecondServlet. In this case, you don't need to worry about the paths of both servlets.

Another option is to do it the harder way by passing the following String to the getRequestDispatcher of ServletRequest:

```
"/servlet/SecondServlet"
```

If you are to use the getRequestDispatcher from the ServletContext, you must pass "/VirtualDir/servlet/SecondServlet" as the path argument, such as the following:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = getServletContext().getRequestDispatcher("/servlet/SecondServlet");
    rd.include(request, response);
}
or
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = getServletContext().getRequestDispatcher("/servlet/SecondServlet");
    rd.forward(request, response);
}
```

To use the getNamedDispatcher method, your code would become

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = getServletContext().getNamedDispatcher("SecondServlet");
    rd.include(request, response);
}
or
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher rd = getServletContext().getNamedDispatcher("SecondServlet");
    rd.forward(request, response);
}
```

Of course, when you use the getNamedDispatcher method, you must register the second servlet in your deployment descriptor. Here is an example:

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>FirstServlet</servlet-class>
</servlet>
```

```
<servlet>
  <servlet-name>SecondServlet</servlet-name>
  <servlet-class>SecondServlet</servlet-class>
</servlet>
</web-app>
```

If you are including from a `doPost` method, the `doPost` method of the second servlet will be invoked. If including from a `doGet` method, the `doGet` method of the second servlet will be called.

Warning! If you change the resource included in your servlet, you need to restart Tomcat for the change to take effect. This is required because the included servlet is never invoked directly. After the included servlet is loaded, its timestamp is never compared again. The following sections give you a closer look at the use of the `RequestDispatcher` interface.

Servlet Chaining:

In the olden days (which are not so long ago, of course), servlet chaining was the technique used to perform what `RequestDispatcher` can do. Servlet chaining is not part of the J2EE specification, however, and its use is dependent on specific servlet containers. You may still find this term in old literature on servlets.

Including Other Resources

On many occasions, you may want to include other resources inside your servlet. For example, you may have a collection of JavaScript functions that you want to include in the response to the user. Separating non-servlet content makes sure that modularity is maintained. In this case, a JavaScript programmer can work independently of the servlet programmer. The page containing the JavaScript functions can then be included using the `include` method of a `RequestDispatcher`.

Another time you may want to include other resources in your servlet might be when you want to include the output of a servlet whose output is a link to a randomly selected advertisement banner. By separating into a separate servlet the code that selects the banner, the same code can be included in more than one servlet, and formatting can be done by modifying the included servlet solely.

The `include` method of the `RequestDispatcher` interface may be called at any time. The target servlet has access to all aspects of the `request` object, but can only write information to the `ServletOutputStream` or `Writer` object of the `response` object. The target servlet also can commit a response by either writing content past the end of the response buffer or explicitly calling the `flush` method of the `ServletResponse` interface. The included servlet cannot set headers or call any method that affects the header of the response.

When a servlet is being called from within an `include` method, it is sometimes necessary for that servlet to know the path by which it was invoked. The following request attributes are set and

accessible from the included servlet via the `getAttribute` method on the request object:

- `javax.servlet.include.request_uri`

- `javax.servlet.include.context_path`
- `javax.servlet.include.servlet_path`
- `javax.servlet.include.path_info`
- `javax.servlet.include.query_string`

These attributes are not set if the included servlet was obtained by using the `getNamedDispatcher` method.

Including Static Content

Sometimes you need to include static content, such as HTML pages or image files that are prepared by a web graphic designer. You can do this by using the same technique for including dynamic resources that you've been reading about in this chapter.

The following example shows a servlet named `FirstServlet` that includes an HTML file named `AdBanner.html`. The servlet class file is located in the `WEB-INF\classes` directory, whereas the `AdBanner.html` file, like other HTML files, resides in the application directory. In other words, using the `myApp` application, the `AdBanner.html` file resides in the `myApp` directory, whereas the servlet class file is in `myApp/WEB-INF/classes` directory.

Including Static Content

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class FirstServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        RequestDispatcher rd = request.getRequestDispatcher("/AdBanner.html");
        rd.include(request, response);
    }
}
```

The AdBanner.html File

```
<HTML>
<HEAD>
<TITLE>Banner</TITLE>
</HEAD>
<BODY>
<IMG SRC=images/banner.jpg>
</BODY>
</HTML>
```

Including Another Servlet

The second example shows a servlet (FirstServlet) that includes another servlet (SecondServlet). The second servlet simply sends the included request parameter to the user. Try this example yourself & see the output.

FirstServlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FirstServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Included Request Parameters</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<B>Included Request Parameters</B><BR>");
        RequestDispatcher rd = request.getRequestDispatcher("/servlet/SecondServlet?name=budi");
        rd.include(request, response);
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

SecondServlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class SecondServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration enum = request.getAttributeNames();
```

```

        while (enum.hasMoreElements()) {
            String attributeName = (String) enum.nextElement();
            out.println(attributeName + ":" + request.getAttribute(attributeName) + "<BR>");
        }
    }
}

```

Forwarding Processing Control

Unlike the include method, the forward method of the RequestDispatcher interface may be called only by the calling servlet if no output has been committed to the client. If output exists in the response buffer that has not been committed, the buffer must be cleared before the target servlet's service method is called. If the response has been committed prior to calling the forward method, an IllegalStateException will be thrown. For example, the servlet given below will raise an error because the flushBuffer method is called before the forward method.

Forwarding Control After the Buffer Is Flushed

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FirstServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Included Request Parameters</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<B>Included Request Parameters</B><BR>");
        response.flushBuffer();
        RequestDispatcher rd = request.getRequestDispatcher("/servlet/SecondServlet");
        rd.forward(request, response);
        out.println("asdfaf</BODY>");
        out.println("</HTML>");
    }
}

```

The forward method also can be used to forward the request to a static content. Again, the flushBuffer method must not be called beforehand.

The forward method is also a good replacement for the sendRedirect method of the HttpServletResponse interface. You may recall that with the sendRedirect method there is a round

trip to the client. If you are redirecting the user to a servlet or a page in the current application, you can use the forward method instead. There is no round trip to the browser when using the forward method; therefore, this gives the user a faster response. The following example rewrites the Login servlet that you developed previously. Instead of using the sendRedirect method, the servlet uses a RequestDispatcher to forward the request to the WelcomeServlet servlet when the login was successful.

The LoginServlet Servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class LoginServlet extends HttpServlet {
    private void sendLoginForm(HttpServletRequest response, boolean withErrorMessage)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Login</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");

        if (withErrorMessage)
            out.println("Login failed. Please try again.<BR>");
        out.println("<BR>Please enter your user name and password.");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<BR>User Name: <INPUT TYPE=TEXT NAME=userName>");
        out.println("<BR>Password: <INPUT TYPE=PASSWORD NAME=password>");
        out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
        out.println("</FORM>");
        out.println("</BODY>");
        out.println("</HTML>");
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        sendLoginForm(response, false);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String userName = request.getParameter("userName");
```

```

String password = request.getParameter("password");
if (userName!=null && password!=null &&      userName.equals("jamesb") &&
password.equals("007")) {
    RequestDispatcher rd = request.getRequestDispatcher("WelcomeServlet");
    rd.forward(request, response);
}
else {
    sendLoginForm(response, true);
}
}
}
}

```

The WelcomeServlet Servlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class WelcomeServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,      HttpServletResponse response)
throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Welcome</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<P>Welcome to the Bulbul's and Boni's Web Site.</P>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}

```

Advanced Servlets Part - 2

Session Management

The Hypertext Transfer Protocol (HTTP) is the network protocol that web servers and client browsers use to communicate with each other. HTTP is the language of the web.

HTTP connections are initiated by a client browser that sends an HTTP request. The web server then responds with an HTTP response and closes the connection. If the same client requests another resource from the server, it must open another HTTP connection to the server. The server always closes the connection as soon as it sends the response, whether or not the browser user needs some other resource from the server.

This process is similar to a telephone conversation in which the receiver always hangs up after responding to the last remark/question from the caller. For example, a call goes something like this:

Caller dials. Caller gets connected.

Caller: "Hi, good morning."

Receiver: "Good morning."

Receiver hangs up.

Caller dials again. Caller gets connected.

Caller: "May I speak to Dr. Zeus, please?"

Receiver: "Sure."

Receiver hangs up.

Caller dials again, and so on, and so on.

Putting this in a web perspective, because the web server always disconnects after it responds to a request, the web server does not know whether a request comes from a user who has just requested the first page or from a user who has requested nine other pages before. As such, HTTP is said to be stateless.

Being stateless has huge implications. Consider, for example, a user who is shopping at an online store. As usual, the process starts with the user searching for a product. If the product is found, the user then enters the quantity of that product into the shopping cart form and submits it to the server. But, the user is not yet checking out—she still wants to buy something else. So she searches the catalog again for the second product. The first product order has now been lost, however, because the previous connection was closed and the web server does not remember anything about the previous connection. To get around such problems, we use some techniques to manage the session.

Session management, also called session tracking, goes beyond simply remembering a user who has successfully logged in. Anything that makes the application remember information that has been entered or requested by the user can be considered session management. Session management does not change the nature of HTTP statelessness—it simply provides a way around it.

By principle, you manage a user's session by performing the following to servlets/pages that need to remember a user's state:

1. When the user requests a servlet, in addition to sending the response, the servlet also sends a token or an identifier.
2. If the user does not come back with the next request for the same or a different servlet, that is fine. If the user does come back, the token or identifier is sent back to the server. Upon encountering the token, the next servlet should recognize the identifier and can do a certain action based on the token. When the servlet responds to the request, it also sends the same or a different token. This goes on and on with all the servlets that need to remember a user's session.

You will use four techniques for session management. They operate based on the same principle, although what is passed and how it is passed is different from one to another. The techniques are as follows:

- URL rewriting
- Hidden fields
- Cookies
- Session objects

Which technique you use depends on what you need to do in your application.

URL Rewriting

With URL rewriting, you append a token or identifier to the URL of the next servlet or the next resource. You can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&...

A name and a value is separated using an equal sign (=); a parameter name/value pair is separated from another parameter name/value pair using the ampersand (&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a servlet, you can use the `HttpServletRequest` interface's `getParameter` method to obtain a parameter value. For instance, to obtain the value of the second parameter, you write the following:

```
request.getParameter(name2);
```

The use of URL rewriting is easy. When using this technique, however, you need to consider several things:

- The number of characters that can be passed in a URL is limited. Typically, a browser can pass up to 2,000 characters.
- The value that you pass can be seen in the URL. Sometimes this is not desirable. For example, some people prefer their password not to appear on the URL.
- You need to encode certain characters—such as & and ? characters and white spaces—that you append to a URL.

Hidden Fields

Another technique for managing user sessions is by passing a token as the value for an HTML hidden field. Unlike the URL rewriting, the value does not show on the URL but can still be read by viewing the HTML source code. Although this method also is easy to use, an HTML form is always required.

Cookies

The third technique that you can use to manage user sessions is by using cookies. A cookie is a small piece of information that is passed back and forth in the HTTP request and response. Even though a cookie can be created on the client side using some scripting language such as JavaScript, it is usually created by a server resource, such as a servlet. The cookie sent by a servlet to the client will be passed back to the server when the client requests another page from the same application.

Cookies were first specified by Netscape and are now part of the Internet standard as specified in RFC 2109: The HTTP State Management Mechanism. Cookies are transferred to and from the client in the HTTP headers.

In servlet programming, a cookie is represented by the `Cookie` class in the `javax.servlet.http` package. You can create a cookie by calling the `Cookie` class constructor and passing two `String` objects: the name and value of the cookie. For instance, the following code creates a cookie object called `c1`. The cookie has the name "myCookie" and a value of "secret":

```
Cookie c1 = new Cookie("myCookie", "secret");
```

You then can add the cookie to the HTTP response using the `addCookie` method of the `HttpServletResponse` interface:

```
response.addCookie(c1);
```

Note that because cookies are carried in the request and response headers, you must not add a cookie after an output has been written to the `HttpServletResponse` object. Otherwise, an exception will be thrown.

The following example shows how you can create two cookies called `userName` and `password` and illustrates how those cookies are transferred back to the server.

```
Cookie c1 = new Cookie("userName", "Helen");
Cookie c2 = new Cookie("password", "Keppler"); response.addCookie(c1);
response.addCookie(c2);
```

To retrieve cookies, you use the `getCookies` method of the `HttpServletRequest` interface. This method returns a `Cookie` array containing all cookies in the request. It is your responsibility to loop through the array to get the cookie you want, as follows:

```
Cookie[] cookies = request.getCookies();
int length = cookies.length;
for (int i=0; i<length; i++) {
    Cookie cookie = cookies[i];
    out.println("<B>Cookie Name:</B> " + cookie.getName() + "<BR>");
    out.println("<B>Cookie Value:</B> " + cookie.getValue() + "<BR>");
}
```

Persisting Cookies

The cookies `c1` and `c2`, created above last as long as the browser is open. When the browser is closed, the cookies are deleted. You can choose to persist cookies so that they last longer. The `javax.servlet.http.Cookie` class has the `setMaxAge` method that sets the maximum age of the cookie in seconds.

Checking Whether Cookie Setting Is On

All the cookie-related examples assume that the user browser's cookie setting is on. Even though browsers leave the factory with this setting on, the user can turn this off. One approach to solving this problem is to send a warning message if the application does not work as expected. The other option is to check this setting automatically. This option is explained below with an example of a servlet called `CheckCookieServlet`.

What the servlet does is simple enough. It sends a response that forces the browser to come back for the second time. With the first response, it sends a cookie. When the browser comes back for the second time, the servlet checks whether the request carries the cookie sent previously. If the cookie is there, it can be concluded that the browser setting for cookies is on. Otherwise, it could be that the user is using a very old browser that does not recognize cookies at all, or the cookie support for that browser is off.

```

CheckCookieServlet
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class CheckCookieServlet extends HttpServlet { /*Process the HTTP Get request*/
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        if(request.getParameter("flag")==null) { // the first request
            Cookie cookie = new Cookie("browserSetting", "on");
            response.addCookie(cookie);
            String nextUrl = request.getRequestURI() + "?flag=1";
            out.println("<META HTTP-EQUIV=Refresh CONTENT=0;URL=" + nextUrl + ">"); }
        else { // the second request
            Cookie[] cookies = request.getCookies();
            if(cookies!=null) {
                int length = cookies.length;
                boolean cookieFound = false;
                for (int i=0; i<length; i++) {
                    Cookie cookie = cookies[i];
                    if(cookie.getName().equals("browserSetting") && cookie.getValue().equals("on")) {
                        cookieFound = true;
                        break;
                    }
                }
                if(cookieFound) {
                    out.println("Your browser's cookie setting is on."); }
                else {
                    out.println("Your browser does not support cookies or" + " the cookie setting is off."); }
            }
        }
    /*Process the HTTP Post request*/
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response); }
}

```

Session Objects

Of the four techniques for session management covered in this chapter, the Session object, represented by the javax.servlet.http.HttpSession interface, is the easiest to use and the most powerful.

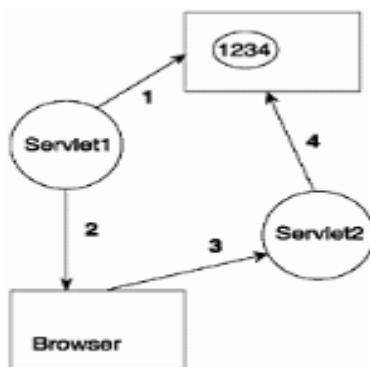
For each user, the servlet can create an HttpSession object that is associated with that user only and can only be accessed by that particular user. The HttpSession object acts like a Hashtable into which you can store any number of key/object pairs. The HttpSession object is accessible from other servlets in the same application. To retrieve an object previously stored, you need only to pass the key.

An HttpSession object uses a cookie or URL rewriting to send a token to the client. If cookies are used to convey session identifiers, the client browsers are required to accept cookies.

Unlike previous techniques, however, the server does not send any value. What it sends is simply a unique number called the session identifier. This session identifier is used to associate a user with a Session object in the server. Therefore, if there are 10 simultaneous users, 10 Session objects will be created in the server and each user can access only his or her own HttpSession object.

The way an HttpSession object is created for a user and retrieved in the next requests is illustrated in the following figure.

How session tracking works with the HttpSession object.



Above figure shows that there are four steps in session tracking using the HttpSession object:

1. An HttpSession object is created by a servlet called Servlet1. A session identifier is generated for this HttpSession object. In this example, the session identifier is 1234, but in reality, the servlet container will generate a longer random number that is guaranteed to be unique. The HttpSession object then is stored in the server and is associated with the generated session identifier. Also the programmer can store values immediately after creating an HttpSession.

2. In the response, the servlet sends the session identifier to the client browser.

3. When the client browser requests another resource in the same application, such as Servlet2, the session identifier is sent back to the server and passed to Servlet2 in the javax.servlet.http.HttpServletRequest object.

4. For Servlet2 to have access to the HttpSession object for this particular client, it uses the getSession method of the javax.servlet.http.HttpServletRequest interface. This method automatically retrieves the session identifier from the request and obtains the HttpSession object associated with the session identifier. **Note:** What if the user never comes back after an HttpSession object is created? Then the servlet container waits for a certain period of time and removes that HttpSession object. One worry about using Session objects is scalability. In some servlet containers, Session objects are stored in memory, and as the number of users exceeds a certain limit, the server eventually runs out of memory.

One solution to this memory problem when using Session objects is to save Session objects to the database or disk. If you are using Tomcat 4 or above, however, your Session objects will be moved to secondary storage once the number of Session objects has exceeded some value.

The getSession method of the javax.servlet.http.HttpServletRequest interface has two overloads. They are as follows:

- *HttpSession getSession()*
- *HttpSession getSession(boolean create)*

The first overload returns the current session associated with this request, or if the request does not have a session identifier, it creates a new one.

The second overload returns the HttpSession object associated with this request if there is a valid session identifier in the request. If no valid session identifier is found in the request, whether a new HttpSession object is created depends on the create value. If the value is true, a new HttpSession object is created if no valid session identifier is found in the request. Otherwise, the getSession method will return null.

Now that you know how session management works using the HttpSession object, let's have a look at the javax.servlet.http.HttpSession interface in more detail.

The javax.servlet.http.HttpSession Interface

This interface has the following methods:

- *getAttribute*
- *getAttributeNames*
- *getCreationTime*
- *getId*
- *getLastAccessedTime*
- *getMaxInactiveInterval*
- *getServletContext*

- `getServletContext`
- `getValue`
- `getValueNames`
- `invalidate`
- `isNew`
- `putValue`
- `removeAttribute`
- `removeValue`
- `setAttribute`
- `setMaxInactiveInterval`

Each of the methods is discussed below.

getAttribute

This method retrieves an attribute from the HttpSession object. The return value is an object of type Object; therefore you may have to downcast the attribute to its original type. To retrieve an attribute, you pass the name associated with the attribute. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The signature for this method is as follows:

```
public Object getAttribute(String name) throws IllegalStateException
```

getAttributeNames

The `getAttributeNames` method returns a `java.util.Enumeration` containing all attribute names in the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The signature is as follows:

```
public java.util.Enumeration getAttributeNames() throws IllegalStateException
```

getCreationTime

The `getCreationTime` method returns the time that the HttpSession object was created, in milliseconds since January 1, 1970 00:00:00 GMT. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The signature for this method is as follows:

```
public long getCreationTime() throws IllegalStateException
```

getId

The `getID` method returns the session identifier. The signature for this method is as follows:

```
public String getId()
```

getLastAccessedTime

The getLastAccessedTime method returns the time the HttpSession object was last accessed by the client. The return value is the number of milliseconds lapsed since January 1, 1970 00:00:00 GMT. The following is the method signature:

```
public long getLastAccessedTime()
```

getMaxInactiveInterval

The getMaxInactiveInterval method returns the number of seconds the HttpSession object will be retained by the servlet container after it is last accessed before being removed. The signature of this method is as follows:

```
public int getMaxInactiveInterval()
```

getServletContext

The getServletContext method returns the javax.servlet.ServletContext object the HttpSession object belongs to. The signature is as follows:

```
public javax.servlet.ServletContext getServletContext
```

getSessionContext

This method is deprecated.

getValue

This method is deprecated.

getValueNames

This method is deprecated.

invalidate

The invalidate method invalidates the HttpSession object and unbinds any object bound to it. This method throws an IllegalStateException if this method is called upon an already invalidated HttpSession object. The signature is as follows:

```
public void invalidate() throws IllegalStateException
```

isNew

The isNew method indicates whether the HttpSession object was created with this request and the client has not yet joined the session tracking. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. Its signature is as follows:

```
public boolean isNew() throws IllegalStateException
```

putValue

This method is deprecated.

removeAttribute

The removeAttribute method removes an attribute bound to this HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.

Its signature is as follows:

```
public void removeAttribute(String name) throws IllegalStateException
```

removeValue

This method is deprecated.

setAttribute

The setAttribute method adds a name/attribute pair to the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The method has the following signature:

```
public void setAttribute(String name, Object attribute) throws IllegalStateException
```

setMaxInactiveInterval

The setMaxInactiveInterval method sets the number of seconds from the time the HttpSession object is accessed the servlet container will wait before removing the HttpSession object. The signature is as follows:

```
public void setMaxInactiveInterval(int interval)  
Passing a negative number to this method will make this HttpSession object never expire.
```

Session Tracking with URL-Rewriting

As mentioned previously, by default a session identifier is sent to the client browser and back to the server using a cookie. This means that the client browser must have its cookie support enabled. Therefore, you need to perform browser cookie testing as explained in the previous section.

Alternatively, you can avoid the use of cookies by sending session identifiers in the URL using the URL-rewriting technique. Fortunately, the Servlet API provides an easy way to append session identifier to the URL—using the encodeURL method of the javax.servlet.http.HttpServletResponse interface. To use the URL-rewriting technique, pass any URL referenced to in a servlet into the encodeURL method of the javax.servlet.http.HttpServletResponse interface. For example, the following code:

```
out.println("<a href=Servlet2>Click here</a>");
```

must be changed into the following:

```
out.println("<a href=" + response.encodeURL("Servlet2") + ">Click here</a>");
```

This will add the session identifier to the end of the string, resulting in a URL similar to the following:

`http://localhost:8080/myApp/servlet/Servlet2;jsessionid=AB348989283429489234`

At where the number after jsessionid= is the session identifier.

Note: One disadvantage of using URL rewriting as opposed to cookies in sending your session identifiers is that URL rewriting does not survive static pages. For example, if the user has to browse through some HTML page during his or her session, the session identifier will be lost.

The not-so-practical solution is of course to convert all static pages into servlets or JSP pages.

Knowing Which Technique to Use

Having learned the four techniques for managing user sessions, you may be wondering which one you should choose to implement.

Clearly, using Session objects is the easiest and you should use this if your servlet container supports swapping Session objects from memory to secondary storage. If you are using Tomcat 4 or above, this feature is available to you.

One concern when using the Session objects is whether cookie support is enabled in the user browser. If it is, you have two options:

- You can test the cookie support setting by using the technique described in the previous section.
- You can use URL-rewriting.

Appending your session identifier to the URL is a good technique, even though this creates some additional work for the programmer. However, this relieves you of having to rely on cookies.

Using cookies is not as flexible as using Session objects. However, cookies are the way to go if you don't want your server to store any client-related information or if you want the client information to persist when the browser is closed.

Finally, hidden fields are probably the least-often-used technique. If you need to split a form into several smaller ones, however, using hidden fields is the cheapest and most efficient method. You don't need to consume server resources to temporarily store the values from the previous forms, and you don't need to rely on cookies. It's suggested hidden fields over Session objects, URL-rewriting, or cookies in this case.

Advanced Servlets Part - 3

Application and Session Events

With Servlet 2.3 API Specification, a new feature was introduced, that has to do with both the ServletContext and HttpSession objects. This new feature supports application and session-level events. If you use Tomcat 4 or higher, this new feature is available to you.

The concept is very simple, so let's take an overview.

In relation to the ServletContext object, the Servlet 2.3 API or higher enables you to get notifications when the ServletContext object is created and destroyed, or when an attribute (a key/object pair) is created, removed, or replaced. The moments these things occur have now become Java events that you can listen to.

The question is, why would you want to know when these things happen? Surely you can live without these events; however, they can be very useful because you can write code that gets called automatically when one of the events happens. For example, you know that the ServletContext object is created by the servlet container when it initializes. You can therefore do certain things in response to this event, such as loading a JDBC driver or creating a database connection object. Or, you probably want to initialize a connection URL and store it as a variable in the ServletContext object so that it is accessible from all the servlets in the web application.

For events that get triggered when the ServletContext is destroyed, you can write code that does some cleanup, such as closing files or ending a database connection.

The same applies to events related to ContextServlet's attributes. For example, you can make a persistent page counter whose value gets written to a file every time the value is changed. Assuming that you store this counter value as an attribute in the ServletContext object, you can write some I/O code that automatically executes when the ServletContext's attribute gets replaced, thus allowing for an accurate count.

Additionally, with an HttpSession object, you also can get notified when an HttpSession object is created or invalidated or when an attribute of a HttpSession object is added, removed, or replaced.

Listening to Application Events

At the application level, the javax.servlet package provides two listener interfaces that support event notifications for state changes in the ServletContext object: the ServletContextListener interface and the ServletContextAttributesListener interface.

The ServletContextListener Interface

You can use the ServletContextListener interface to listen to the ServletContext life cycle events. Its signature is given as follows:

```
public interface ServletContextListener extends java.util.EventListener
```

Your listener class must implement this interface to listen to ServletContext life cycle events. The ServletContextListener interface has two methods: contextInitialized and contextDestroyed. The signatures for these methods are the following:

```
public void contextInitialized(ServletContextEvent sce)  
public void contextDestroyed(ServletContextEvent sce)
```

The contextInitialized method is called when the web application is ready to service requests. The method is called automatically by the servlet container when its own initialization process is finished. You can write code that needs to get executed when the application initializes, such as loading a JDBC driver, creating a database Connection object, or assigning initialization values to global variables.

The contextDestroyed method is invoked when the servlet context is about to be shut down. You can use this method to write code that needs to run when the application shuts down, such as closing a database connection or writing to the log.

Utilizing the contextInitialized method is similar to writing code in a servlet's init() method, and the contextDestroyed method has a similar effect as a servlet's destroy() method. However, using application events make the codes available throughout the whole application, not only from inside a servlet.

The ServletContextEvent Class

As you have seen, both methods of the ServletContextListener interface pass a ServletContextEvent class. This class has the following signature:

```
public class ServletContextEvent extends java.util.EventObject
```

The ServletContextEvent has only one method: getServletContext. This method returns the ServletContext that is changed.

Deployment Descriptor

To use application events and include a servlet listener class, you must tell the servlet container by registering the listener class in the deployment descriptor. The <web-app> element must contain a <listener> element like the following:

```
<web-app>
  <listener>
    <listener-class>AppLifeCycleEvent</listener-class>
  </listener>
</web-app>
```

The <listener> element must come before the <servlet> part. AppLifeCycleEvent Class

The following is a simple listener class that listens to the life cycle events of the ServletContext. It simply prints the string "Application initialized" when the ServletContext is initialized and "Application destroyed" when the ServletContext is destroyed. The code for the program is given below.

The AppLifeCycleEvent Class

```
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;

public class AppLifeCycleEvent implements ServletContextListener {
  public void contextInitialized(ServletContextEvent cse) {
    System.out.println("Application initialized");
  }
  public void contextDestroyed(ServletContextEvent cse) {
    System.out.println("Application shut down");
  }
}
```

For the AppLifeCycleEvent class to work, you must register it in the deployment descriptor. The deployment descriptor is given below.

The Deployment Descriptor for the AppLifeCycleEvent Class

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <listener>
    <listener-class>AppLifeCycleEvent</listener-class>
  </listener>
</web-app>
```

Now, restart Tomcat and watch the console window. You should be able to see the string "Application initialized" at the console.

If you want, you can have more than one listener class. To do this, list all the listener classes in the deployment descriptor, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<listener>
<listener-class> AppLifeCycleEvent1 </listener-class>
</listener>
<listener>
<listener-class> AppLifeCycleEvent2 </listener-class>
</listener>
</web-app>
```

With a deployment descriptor like this, when Tomcat starts, it will call the contextInitialized method in the AppLifeCycleEvent1 class and then call the same method in the AppLifeCycleEvent2 class.

Another Example: Loading a JDBC Driver and Setting a ServletContext Attribute

The following example uses the contextInitialized method in the ServletContextListener interface to load a JDBC driver and set a ServletContext attribute named "dbUrl" with a value of "jdbc:mysql//Fred". Loading the JDBC driver here makes it available for the next access to the database.

The example has two classes: a listener class called AppLifeCycleEventDemo and a servlet named ApplicationEventDemoServlet. The listener class provides a method (contextInitialized) that will be called automatically when the ServletContext object is initialized. This is where you put the code that loads the JDBC driver and sets a ServletContext attribute. The servlet is used to display the attribute value.

AppLifeCycleEventDemo

```
import javax.servlet.ServletContext;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;

public class AppLifeCycleEventDemo implements ServletContextListener {
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("Initializing Application ..."); // Load the JDBC driver
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
```

```

}

catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}

// Get the ServletContext object
ServletContext servletContext = sce.getServletContext();
// Set a ServletContext attribute
servletContext.setAttribute("dbUrl", "jdbc:mysql://Fred");
System.out.println("Application initialized"); }

public void contextDestroyed(ServletContextEvent cse) {
    System.out.println("Application shut down");
}
}

```

Note that before you can set a ServletContext attribute, you first need to obtain the ServletContext object using the getServletContext method of the ServletContextEvent class. Following is how we got the servletcontext reference.

```
ServletContext servletContext = sce.getServletContext();
```

In the servlet, to get an attribute from the ServletContext object, you need first to obtain the ServletContext object. In a servlet that extends the javax.servlet.http.HttpServlet class, you can use the getServletContext method to achieve this, as demonstrated in the following code.

The ApplicationEventDemoServlet Servlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ApplicationEventDemoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Application Event Demo Servlet</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("Your database connection is ");
        // get the ServletContext object
        ServletContext servletContext = getServletContext(); // display the "dbUrl" attribute
        out.println(servletContext.getAttribute("dbUrl"));
    }
}

```

```

        out.println("</BODY>");
        out.println("</HTML>");
    }
}

```

The deployment descriptor you need for the application to work, is given below.

The Deployment Descriptor for the Example

```

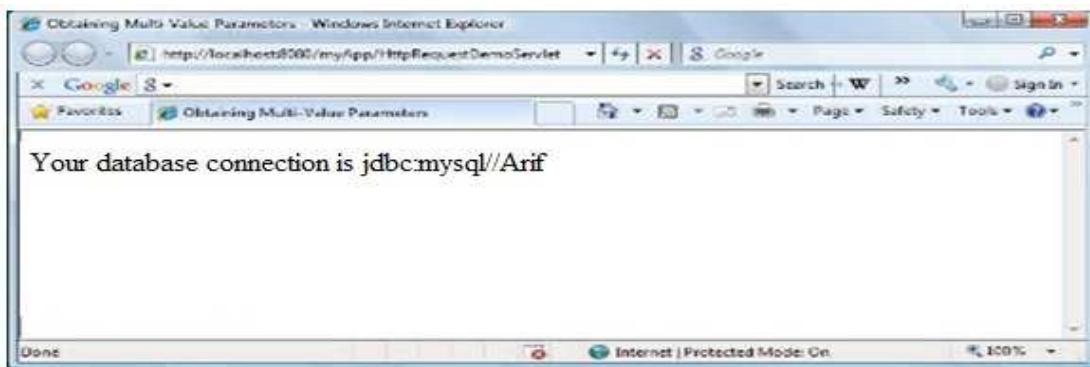
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-
app_2_3.dtd">

<web-app>
<listener>
<listener-class> AppLifeCycleEventDemo </listener-class>
</listener>
<servlet>
<servlet-name> ApplicationEventDemo </servlet-name>
<servlet-class> ApplicationEventDemoServlet </servlet-class>
</servlet>
</web-app>

```

After you compile both classes, restart Tomcat and direct your browser to the servlet URL. You should see something similar to following figure.

Obtaining an attribute value set when the application initializes.



Listening to ServletContextAttributeListener

In addition to a life cycle listener, you also can implement the `ServletContextAttributeListener` class to be notified when any attribute is added to the `ServletContext` or if any of the `ServletContext`'s attributes are changed or removed. Implementing this interface, you must provide implementations

for its three methods: attributeAdded, attributeRemoved, and attributeReplace. The signatures for those methods are as follows:

```
public void attributeAdded(ServletContextAttributeEvent scae)  
public void attributeRemoved(ServletContextAttributeEvent scae)  
public void attributeReplaced(ServletContextAttributeEvent scae)
```

The attributeAdded method is called when a new attribute is added to the ServletContext object. The attributeRemove method is called when an attribute is removed from the ServletContext object, and the attributeReplaced method is invoked when an attribute in the ServletContext object is replaced.

Note that attribute changes may occur concurrently. It is your responsibility to ensure that accesses to a common resource are synchronized.

Another Example: PageCounterServlet

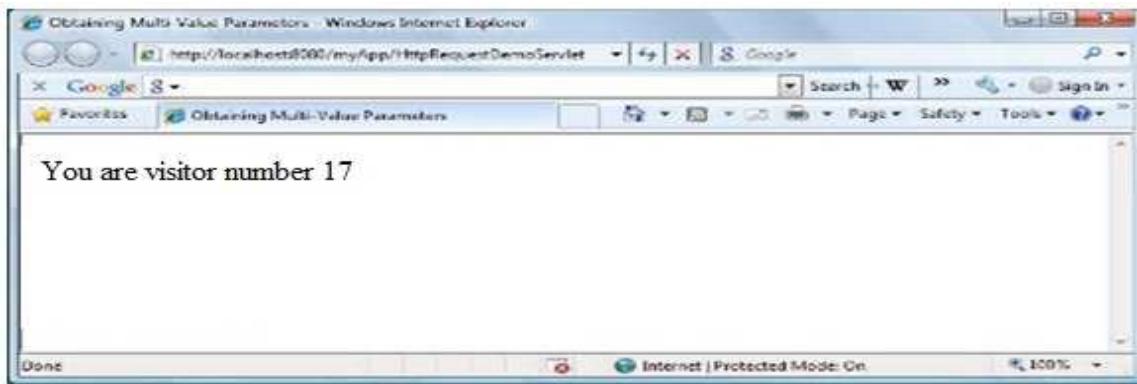
As another example, consider the following application that provides a page counter for a servlet. The servlet increments a counter every time it is requested, thus the name page counter. The value of the counter is stored as an attribute of the ServletContext object so that it can be accessed from any servlet in the application. However, if the servlet container crashes, you will not lose this value because the value is also written to a text file called counter.txt. Clearly, this is a type of application you can use to measure the popularity of your site.

The application consists of two classes. The first is a listener class named AppAttributeEventDemo and the second is a servlet called PageCounterServlet. The AppAttributeEventDemo implements both ServletContextListener interface and ServletContextAttributeListener interface. In the contextInitialized method, you write the code that will read the counter value from the counter.txt file and use this value to initialize a ServletContext attribute named pageCounter. As in the previous example, you need to obtain the ServletContext object from the ServletContextEvent object passed to the contextInitialized method.

Every time the PageCounterServlet servlet is requested, it will increment the value of pageCounter. When this happens, the attributeReplaced method of the listener class will be triggered. In this method, you write the code that writes the value of pageCounter to the counter.txt file.

Because the servlet can be called simultaneously by multiple users, the method writeCounter, called from the attributeReplaced method, is synchronized. Here is where you write the code to write pageCounter to the file.

The listener class and the PageCounterServlet are given below. After you compile these two classes, open your browser and type the URL for the servlet. You should see something similar to following figure.



The AppAttributeEventDemo class

```
import javax.servlet.ServletContext;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextAttributeEvent;
import java.io.*;  
  
public class AppAttributeEventDemo  
    implements ServletContextListener, ServletContextAttributeListener {  
    int counter;  
    String counterFilePath = "C:\\counter.txt";  
    public void contextInitialized(ServletContextEvent cse) {  
        try {  
            BufferedReader reader = new BufferedReader(new FileReader(counterFilePath));  
            counter = Integer.parseInt( reader.readLine() );  
            reader.close();  
            System.out.println("Reading " + counter);  
        }  
        catch (Exception e) {  
            System.out.println(e.toString());  
        }  
        ServletContext servletContext = cse.getServletContext();  
        servletContext.setAttribute("pageCounter", Integer.toString(counter));  
        System.out.println("Application initialized");    }  
  
    public void contextDestroyed(ServletContextEvent cse) {  
        System.out.println("Application shut down");    }  
  
    public void attributeAdded(ServletContextAttributeEvent scae) {  
        System.out.println("ServletContext attribute added");    }  
}
```

```

public void attributeRemoved(ServletContextAttributeEvent scae) {
    System.out.println("ServletContext attribute removed");
}

public void attributeReplaced(ServletContextAttributeEvent scae) {
    System.out.println("ServletContext attribute replaced");
    writeCounter(scae);
}

synchronized void writeCounter(ServletContextAttributeEvent scae) {
    ServletContext servletContext = scae.getServletContext();
    counter = Integer.parseInt((String)
        servletContext.getAttribute("pageCounter"));
    try {
        BufferedWriter writer = new BufferedWriter(new FileWriter(counterFilePath));
        writer.write(Integer.toString(counter));
        writer.close();
        System.out.println("Writing");
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
}
}

```

The PageCounterServlet Servlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PageCounterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Page Counter</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        ServletContext servletContext = getServletContext();
        int pageCounter = Integer.parseInt((String)servletContext.getAttribute("pageCounter"));

```

```

pageCounter++;
out.println("You are visitor number " + pageCounter);
servletContext.setAttribute("pageCounter", Integer.toString(pageCounter));
out.println("</BODY>");
out.println("</HTML>");
}
}

```

The deployment descriptor for the example is given below.

The Deployment Descriptor for the PageCounter Example

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<listener>
<listener-class>    AppAttributeEventDemo    </listener-class>
</listener>
<servlet>
<servlet-name>    PageCounter    </servlet-name>
<servlet-class>    PageCounterServlet    </servlet-class>
</servlet>
</web-app>

```

Note that the attributeReplaced also is called when the attributeAdded is triggered.

Listening to HttpSession Events

The javax.servlet.http package provides two interfaces that you can implement to listen to HttpSession events: HttpSessionListener and HttpSessionAttributeListener. The first enables you to listen to a session's life cycle events, that is, the event that gets triggered when an

HttpSession object is created and the event that is raised when an HttpSession object is destroyed. The second interface, HttpSessionAttributeListener, provides events that get raised when an attribute is added to, removed from, or modified in the HttpSession object. The two interfaces are explained next.

The HttpSessionListener Interface

The HttpSessionListener interface has two methods: sessionCreated and sessionDestroyed. The signatures of the two methods are given here:

```

public void sessionCreated(HttpSessionEvent se)
public void sessionDestroyed(HttpSessionEvent se)

```

The sessionCreated method is automatically called when an HttpSession object is created. The sessionDestroyed method is called when an HttpSession object is invalidated. Both methods will be passed in an HttpSessionEvent class that you can use from inside the method.

The HttpSessionEvent class is derived from the java.util.EventObject class. The HttpSessionEvent class defines one new method called getSession that you can use to obtain the HttpSession object that is changed.

An Example: User Counter

The following example is a counter that counts the number of different users currently "in session." It creates an HttpSession object for a user the first time the user requests the servlet. If the user comes back to the same servlet, no HttpSession object will be created the second time. Therefore, the counter is incremented only when an HttpSession object is created. An HttpSession object can also be destroyed, however. When this happens, the counter must be decremented. For the counter to be accessible to all users, it is stored as an attribute in the ServletContext object.

The example has two classes: the listener class and a servlet class that displays the counter value. The listener class is named SessionLifeCycleEventDemo and implements both the ServletContextListener interface and the HttpSessionListener interface. You need the first interface so that you can create a ServletContext attribute and assign it an initial value of 0. As you can guess, you put this code in the contextInitialized method, as in the following:

```
public void contextInitialized(ServletContextEvent sce) {  
    servletContext = sce.getServletContext();  
    servletContext.setAttribute("userCounter", Integer.toString(counter)); }
```

Notice that the ServletContext object is assigned to a class variable servletContext, making the ServletContext object available from anywhere in the class.

The counter must be incremented when an HttpSession is created and decremented when an HttpSession is destroyed. Therefore, you need to provide implementations for both sessionCreated and sessionDestroyed methods, as you see here:

```
public void sessionCreated(HttpSessionEvent hse) {  
    System.out.println("Session created.");  
    incrementUserCounter();  
}  
public void sessionDestroyed(HttpSessionEvent hse) {  
    System.out.println("Session destroyed.");  
    decrementUserCounter();  
}
```

The sessionCreated method calls the synchronized method incrementUserCounter and the sessionDestroyed method calls the synchronized method decrementUserCounter.

The incrementUserCounter method first obtains an attribute called userCounter from the ServletContext object, increments the counter, and stores the counter back to the userCounter attribute.

```

synchronized void incrementUserCounter() {
    counter = Integer.parseInt( (String)servletContext.getAttribute("userCounter"));
    counter++;
    servletContext.setAttribute("userCounter", Integer.toString(counter));
    System.out.println("User Count: " + counter);
}

```

The decrementUserCounter method does the opposite. It first obtains the userCounter attribute from the ServletContext object, decrements the counter, and stores the counter back to the userCounter attribute, as you see here:

```

synchronized void decrementUserCounter() {
    int counter = Integer.parseInt( (String)servletContext.getAttribute("userCounter"));
    counter--;
    servletContext.setAttribute("userCounter", Integer.toString(counter));
    System.out.println("User Count: " + counter);
}

```

The listener class is given below.

The SessionLifeCycleEventDemo Class

```

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;

public class SessionLifeCycleEventDemo
    implements ServletContextListener, HttpSessionListener {
    ServletContext servletContext;
    int counter;
    public void contextInitialized(ServletContextEvent sce) {
        servletContext = sce.getServletContext();
        servletContext.setAttribute("userCounter", Integer.toString(counter));    }

    public void contextDestroyed(ServletContextEvent sce) {    }
    public void sessionCreated(HttpSessionEvent hse) {
        System.out.println("Session created.");
        incrementUserCounter();
    }
}

```

```

public void sessionDestroyed(HttpSessionEvent hse) {
    System.out.println("Session destroyed.");
    decrementUserCounter();
}

synchronized void incrementUserCounter() {
    counter = Integer.parseInt((String)servletContext.getAttribute("userCounter"));
    counter++;
    servletContext.setAttribute("userCounter", Integer.toString(counter));
    System.out.println("User Count: " + counter);  }

synchronized void decrementUserCounter() {
    int counter = Integer.parseInt((String)servletContext.getAttribute("userCounter"));
    counter--;
    servletContext.setAttribute("userCounter", Integer.toString(counter));
    System.out.println("User Count: " + counter);  }
}

```

The second class of the application is the UserCounterServlet. This servlet displays the value of the userCounter ServletContext attribute whenever someone requests the servlet. The code that does this is written in the doGet method.

When the doGet method is invoked, you obtain the ServletContext object by calling the getServletContext method.

```
ServletContext servletContext = getServletContext();
```

Then, the method tries to obtain an HttpSession object from the HttpServletRequest object and creates one if no HttpSession object exists, as follows:

```
HttpSession session = request.getSession(true);
```

Next, you need to get the userCounter attribute from the ServletContext object:

```

int userCounter = 0;
userCounter = Integer.parseInt(
    (String)servletContext.getAttribute("userCounter"));

```

The complete code for the servlet is given below.

The userCounterServlet Servlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class UserCounterServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

```

```

ServletContext servletContext = getServletContext();
HttpSession session = request.getSession(true);
int userCounter = 0;
userCounter = Integer.parseInt((String)servletContext.getAttribute("userCounter"));
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>User Counter</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("There are " + userCounter + " users.");
out.println("</BODY>");
out.println("</HTML>");
}
}

```

For the example to work, you need to create a deployment descriptor as given below.

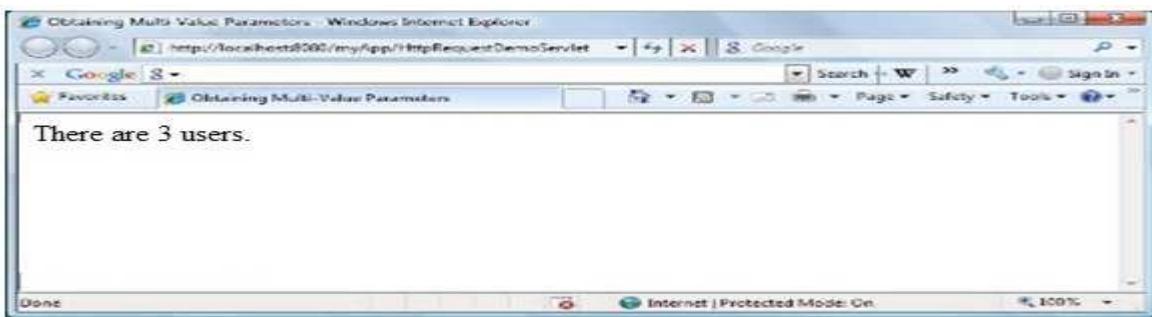
The Deployment Descriptor for the UserCounter Example

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<listener>
<listener-class> SessionLifeCycleEventDemo </listener-class>
</listener>
<servlet>
<servlet-name> UserCounter </servlet-name>
<servlet-class> UserCounterServlet </servlet-class>
</servlet>
</web-app>

```

After you compile the two classes, open your browser and type in the URL for the servlet. You should see something similar to the following.



The HttpSessionAttributeListener Interface

The second session event listener interface is HttpSessionAttributeListener. You can implement this interface if you need to listen to events related to session attributes. The interface provides three methods: attributeAdded, attributeRemoved, and attributeReplaced. The signatures of the three methods are as follows:

```
public void attributeAdded(HttpSessionBindingEvent sbe)  
public void attributeRemoved(HttpSessionBindingEvent sbe)  
public void attributeReplaced(HttpSessionBindingEvent sbe)
```

The attributeAdded method is called when an attribute is added to an HttpSession object. The attributeRemoved and attributeReplaced methods are called when an HttpSession attribute is removed or replaced, respectively.

All the methods receive an HttpSessionBindingEvent object whose class is described next. The HttpSessionBindingEvent is derived from the HttpSessionEvent class so it inherits the getSession method. In addition, the HttpSessionBindingEvent class has two methods: getName and getValue. The signatures of the two methods are as follows:

```
public String getName()  
public Object getValue()
```

The getName method returns the name of the attribute that is bound to an HttpSession or unbound from an HttpSession. The getValue method returns the value of an HttpSession attribute that has been added, removed, or replaced.

Servlet Filtering

Filters were introduced in Servlet 2.3 specification, enabling you to intercept a request before it reaches a resource. In other words, a filter gives you access to the HttpServletRequest and the HttpServletResponse objects before they are passed in to a servlet.

Filters can be very useful. For example, you can write a filter that records all incoming requests and logs the IP addresses of the computers from which the requests originate. You also can use a filter as an encryption and decryption device. Other uses include user authentication, data compression, user input validation, and so on.

For a filter to intercept a request to a servlet, you must declare the filter with a <filter> element in the deployment descriptor and map the filter to the servlet using the <filter-mapping> element. Sometimes you want a filter to work on multiple servlets. You can do this by mapping a filter to a URL pattern so that any request that matches that URL pattern will be filtered.

You also can put a set of filters in a chain. The first filter in the chain will be called first and then pass control to the second filter, and so on. Filter chaining ensures that you can write a filter that does a specific task but adds some functionality in another filter.

Let's start with javax.servlet.Filter, an interface that a filter class must implement. We'll then move on to building various filters and along the way introduces other relevant classes and interfaces.

An Overview of the API

When writing a filter, you basically deal with the following three interfaces in the javax.servlet package:

- Filter
- FilterConfig
- FilterChain

These three interfaces are described in the next sections.

The Filter Interface

javax.servlet.Filter is an interface that you must implement when writing a filter. The life cycle of a filter is represented by this interface's three methods: init, doFilter, and destroy. Their signatures are given here:

```
public void init(FilterConfig filterConfig)
public void doFilter(HttpServletRequest request,
                      HttpServletResponse response, FilterChain chain)
public void destroy()
```

A filter starts its life when its init method is called by the servlet container. The servlet container calls a filter's init method only once, when it finishes instantiating the filter. The servlet container will pass in the FilterConfig object that represents the filter configuration. Normally, you assign this object to an object variable so that the FilterConfig object will be available from other methods. Compare the init method with the init method of the Servlet interface.

The doFilter method is where the filtering is performed. The servlet container calls the doFilter method every time a user requests a resource, such as a servlet, to which the filter is mapped. When doFilter is invoked, the servlet container passes in the HttpServletRequest object, the HttpServletResponse object, and a FilterChain object. The HttpServletRequest and HttpServletResponse objects are the same objects that will get passed to a servlet. You can manipulate these two objects. For example, you can add an attribute to the HttpServletRequest using the setAttribute method, or you can obtain the PrintWriter object of the HttpServletResponse and write to it. The FilterChain object is useful here so that you can pass control to the next resource. It will be explained in more detail when we discuss the FilterChain later in this chapter.

The doFilter method is analogous to the service method of the Servlet interface. The servlet container calls the destroy method to tell the filter that it will be taken out of service. The filter can then do some clean-up, if necessary. For instance, it can close a resource that it opened at initialization. The destroy method in the Filter interface is similar to the destroy method in the Servlet interface.

The FilterConfig Interface

A FilterConfig object represents the configuration for the filter. This object allows you to obtain the ServletContext object and pass initialization values to the filter through its initial parameters, which you define in the deployment descriptor when declaring the filter.

The FilterConfig interface has four methods: getFilterName, getInitParameter, getInitParameterNames, and getServletContext.

The signatures for these methods are as follows:

```
public String getFilterName()  
public String getInitParameter(String parameterName)  
public java.util.Enumeration getInitParameterNames()  
public ServletContext getServletContext()
```

The getFilterName method returns the name of the filter and the getServletContext method returns the ServletContext object. The getInitParameterNames gives you an Enumeration containing all parameter names of the filter. You then can retrieve the value of each individual initial parameter using the getInitParameter, passing the parameter name.

The FilterChain Interface

As mentioned previously, a FilterChain object is passed in by the servlet container to the doFilter method of the filter class. Filters use the FilterChain object to invoke the next filter in the chain, or, if the filter is the last in the chain, to invoke the next resource (servlet). The FilterChain interface has only one method: doFilter, whose signature is given as follows: public void doFilter(HttpServletRequest request, HttpServletResponse response)

Note: Don't confuse this method with the doFilter method of the Filter interface. The latter has three arguments.

You should always call the FilterChain interface's doFilter method to pass control over to the next filter. If you are using only one filter, the doFilter method passes control to the next resource, which can be a servlet you want to filter. Failure to call this method will make the program flow stop.

A Basic Filter

As an example of your first filter, A very basic filter has been presented that does nothing other than show its life cycle by printing a message to the console every time its method is invoked. This filter also declares a FilterConfig object reference called filterConfig. In the init method, you pass the FilterConfig object to this variable. Notice that the filter's doFilter method calls the doFilter method of the FilterChain object at the last line of the method. The code for the filter has been given below.

The BasicFilter Class

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class BasicFilter implements Filter {
    private FilterConfig filterConfig;
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("Filter initialized");
        this.filterConfig = filterConfig;
    }
    public void destroy() {
        System.out.println("Filter destroyed");
        this.filterConfig = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        System.out.println("doFilter");
        chain.doFilter(request, response);
    }
}
```

For the filter to work, you need to decide which servlet or servlets you want to filter and tell the servlet container by declaring the filter in the deployment descriptor using the <filter> element and the <filter-mapping> element. These two elements must come before any <listener> and <servlet> elements. The deployment descriptor for the above example is given below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- Define servlet-mapped and path-mapped filters -->
    <filter>
        <filter-name> Basic Filter </filter-name>
        <filter-class> BasicFilter </filter-class>
    </filter>
```

```

<!-- Define filter mappings for the defined filters -->
<filter-mapping>
    <filter-name> Basic Filter </filter-name>
    <servlet-name> FilteredServlet </servlet-name>
</filter-mapping>
<listener>
    <listener-class> SessionLifeCycleEventDemo </listener-class>
</listener>
<servlet>
    <servlet-name> FilteredServlet </servlet-name>
    <servlet-class> FilteredServlet </servlet-class>
</servlet>
</web-app>

```

You can see that the filter name is Basic Filter and its class is BasicFilter. You also find a <filter-mapping> element that maps the filter to a servlet called FilteredServlet. You need to create a servlet (that does anything) called FilteredServlet to see the example work.

After you compile the filter and servlet classes and modify your deployment descriptor, restart the servlet container. You should see the message sent from the filter's init method in the console.

Open your browser and type the URL to the FilteredServlet servlet. You should see more messages in the console that show how the filter is invoked before the servlet.

If you want to apply the filter to more than one servlet, you need only to repeat the <filter-mapping> element for each servlet. For example, the filter applies to both FilteredServlet and FilteredServlet2 in the following deployment descriptor.

The Deployment Descriptor that Applies the Filter to More than One Servlet

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- Define servlet-mapped and path-mapped filters -->
    <filter>
        <filter-name>Basic Filter</filter-name>
        <filter-class>BasicFilter</filter-class>
    </filter>

    <!-- Define filter mappings for the defined filters -->
    <filter-mapping>
        <filter-name>Basic Filter</filter-name>

```

```

<servlet-name>FilteredServlet</servlet-name>  </filter-mapping>
<filter-mapping>
  <filter-name>Basic Filter</filter-name>
  <servlet-name>FilteredServlet2</servlet-name>  </filter-mapping>

<servlet>
  <servlet-name>FilteredServlet</servlet-name>
  <servlet-class>FilteredServlet</servlet-class>
</servlet>
<servlet>
  <servlet-name>FilteredServlet2</servlet-name>
  <servlet-class>FilteredServlet2</servlet-class>
</servlet>
</web-app>

```

Mapping a Filter with a URL

In addition to mapping a filter to a servlet or a number of servlets, you can map the filter to a URL pattern so all requests that match that pattern will invoke the filter.

Consider a servlet whose URL is similar to the following:

http://localhost:8080/myApp/servlet/FilteredServlet

To map a filter to a URL pattern, you use the `<filter-mapping>` element of your deployment descriptor similar to the following:

```

<!-- Define filter mappings for the defined filters --> <filter-mapping>
<filter-name>Logging Filter</filter-name>
<url-pattern>/servlet/FilteredServlet</url-pattern> </filter-mapping>

```

As an alternative, you can use `/*` to make the filter work for all static and dynamic resources, as follows:

```

<!-- Define filter mappings for the defined filters --> <filter-mapping>
<filter-name>Logging Filter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>

```

A Logging Filter

Here is another simple example of a filter that logs user IP addresses to the log file. The location of the log file is servlet container implementation specific. The filter class is given below.

```

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class LoggingFilter implements Filter {
    private FilterConfig filterConfig = null;
    public void destroy() {

        System.out.println("Filter destroyed");
        this.filterConfig = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("doFilter");

        // Log user's IP address.
        ServletContext servletContext = filterConfig.getServletContext();
        servletContext.log(request.getRemoteHost());
        chain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("Filter initialized");
        this.filterConfig = filterConfig;
    }
}

```

Not much difference exists between this filter and the BasicFilter, except that the doFilter method obtains the ServletContext object and uses its log to record the client IP address obtained from the getRemoteHost method of the HttpServletRequest object. The following code shows this action:

```

// Log user's IP address.
ServletContext servletContext = filterConfig.getServletContext();
servletContext.log(request.getRemoteHost());

```

The deployment descriptor of this example is given below.

The Deployment Descriptor for the LoggingFilter

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <!-- Define servlet-mapped and path-mapped filters -->
    <filter>
        <filter-name>Logging Filter</filter-name>
        <filter-class>LoggingFilter</filter-class>
    </filter>

    <!-- Define filter mappings for the defined filters -->
    <filter-mapping>
        <filter-name>Logging Filter</filter-name>
        <servlet-name>FilteredServlet</servlet-name>
    </filter-mapping>

    <servlet>
        <servlet-name>FilteredServlet</servlet-name>
        <servlet-class>FilteredServlet</servlet-class>
    </servlet>
</web-app>
```

Practically, any servlet named FilteredServlet will invoke the filter. Following is an example of such a servlet.

A Servlet to Be Used with LoggingFilter

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FilteredServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>User Counter</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
```

```

        out.println("IP:" + request.getRemoteHost());
        out.println("</BODY>");
        out.println("</HTML>");
    }
}

```

Filter Configuration

You can pass some initial parameters to a filter in a FilterConfig object that gets passed into the init method of the Filter interface. The initial parameters are declared in the deployment descriptor using the <init-param> element under the <filter> element. For example, the following deployment descriptor describes a filter called MyFilter with two initial parameters: adminPhone and adminEmail.

The Deployment Descriptor with a Filter with Two Initial Parameters

```

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<!-- Define servlet-mapped and path-mapped filters -->

<filter>
<filter-name>MyFilter</filter-name>
<filter-class>MyFilter</filter-class>

<init-param>
    <param-name>adminPhone</param-name>
    <param-value>0414789098</param-value>
</init-param>

<init-param>
    <param-name>adminEmail</param-name>
    <param-value>admin@labsale.com</param-value>
</init-param>

</filter>
</web-app>

```

You then can retrieve the values of the AdminPhone and AdminEmail parameters using the following code, which you should put under the doFilter method of a filter:

```

String adminPhone = filterConfig.getInitParameter("adminPhone");
String adminEmail = filterConfig.getInitParameter("adminEmail");

```

A Filter that Checks User Input

When you receive input from a user, the first thing you should do is to check whether the input is valid. If the input is invalid, you normally send an error message, telling the user that a correct entry is needed.

Sometimes the input is not totally invalid; it only contains leading or trailing empty spaces. In this case, you don't need to send an error message, but you can make the correction yourself; that is, when you ask users to enter their first names into the firstName box in an HTML form, you need to make sure that they type in something like "John" or "George", not "John" or "George" (with blank spaces). Normally in situations like this, you use the trim function of the String class when you obtain a parameter value, as follows:

```
String firstName = request.getParameter("firstName");
if(firstName!=null)
    firstName = firstName.trim();
```

If you have quite a large number of input boxes in the HTML form, however, you have to call the trim method for every parameter. Additionally, you need to do this in every servlet that accepts user input. A filter can help make this task easier.

You can write a filter that trims every parameter value in the HttpServletRequest object before the HttpServletRequest object reaches a servlet. This way, you don't need to trim anything in your servlet. You need to write only one filter to serve all servlets that need this service. In the example that follows, you get a chance to write a filter that trims all parameter values.

To ensure that the filter is able to trim all parameters in a HttpServletRequest object, do not hard-code the name of the parameter. Instead, use the getParameterNames method to obtain an Enumeration containing all the parameter names. Next, you need to loop through the Enumeration to get the parameter values and call the trim method.

You can't change a parameter value, however, which means that it is not possible to trim it directly. A closer look at the HttpServletRequest reveals that you can set its attribute. What you can do is to put trimmed parameter values as attributes. The parameter names become attribute names. Later, in the servlet, instead of retrieving user input from HttpServletRequest parameters, you can obtain the trimmed versions of the input from the HttpServletRequest attributes, as shown here:

```
Enumeration enum = request.getParameterNames();
while (enum.hasMoreElements()) {
    String parameterName = (String) enum.nextElement();
    String parameterValue = request.getParameter(parameterName);
    request.setAttribute(parameterName, parameterValue.trim());
}
```

In your servlet, do the following to get a trimmed input value:

```
request.getAttribute(parameterName);
```

Instead of retrieving a value using the getParameter method of the HttpServletRequest, you use its getAttribute method.

The filter that does this service is called TrimFilter and its code is given below.

The TrimFilter

```
import java.io.*;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import java.util.Enumeration;

public class TrimFilter implements Filter {
    private FilterConfig filterConfig = null;
    public void destroy() {
        System.out.println("Filter destroyed");
        this.filterConfig = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        System.out.println("Filter");

        Enumeration enum = request.getParameterNames();
        while (enum.hasMoreElements()) {
            String parameterName = (String) enum.nextElement();
            String parameterValue = request.getParameter(parameterName);
            request.setAttribute(parameterName, parameterValue.trim());
        }
        chain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("Filter initialized");
        this.filterConfig = filterConfig;
    }
}
```

To illustrate the filter's use, you can write a servlet that does the following:

- Send an HTML form with four input boxes (firstName, lastName, userName, and password) when its doGet method is invoked.
- Display the user input when its doPost method is invoked. The servlet is called TrimFilteredServlet and is given below.

TrimFilteredServlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import com.brainysoftware.java.StringUtil;

public class TrimFilteredServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>User Input Form</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<CENTER>");
        out.println("<BR>Please enter your details.");
        out.println("<BR>");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<TABLE>");
        out.println("<TR>");
        out.println("<TD>First Name:</TD>");
        out.println("<TD><INPUT TYPE=TEXT NAME=firstName></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD>Last Name:</TD>");
        out.println("<TD><INPUT TYPE=TEXT NAME=lastName></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD>User Name:</TD>");
        out.println("<TD><INPUT TYPE=TEXT NAME=userName></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD>Password:</TD>");
```

```

        out.println("<TD><INPUT TYPE=PASSWORD NAME=password></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD ALIGN=RIGHT COLSPAN=2>");
        out.println("<INPUT TYPE=SUBMIT VALUE=Login></TD>");    out.println("</TR>");
        out.println("</TABLE>");
        out.println("</FORM>");
        out.println("</CENTER>");
        out.println("</BODY>");
        out.println("</HTML>");
    }

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String firstName = (String) request.getAttribute("firstName");
String lastName = (String) request.getAttribute("lastName");
String userName = (String) request.getAttribute("userName");
String password = request.getParameter("password");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>Displaying Values</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("<CENTER>");
out.println("Here are your details.");
out.println("<TABLE>");
out.println("<TR>");
out.println("<TD>First Name:</TD>");
out.println("<TD>" + StringUtil(firstName) + "</TD>");
out.println("</TR>");    out.println("<TR>");
out.println("<TD>Last Name:</TD>");
out.println("<TD>" + StringUtil(lastName) + "</TD>");
out.println("</TR>");    out.println("<TR>");
out.println("<TD>User Name:</TD>");
out.println("<TD>" + StringUtil(userName) + "</TD>");
out.println("</TR>");    out.println("<TR>");
out.println("<TD>Password:</TD>");
out.println("<TD>" + StringUtil(password) + "</TD>");
out.println("</TABLE>");

}

```

```

    out.println("</CENTER>");
    out.println("</BODY>");
    out.println("</HTML>");
}
}

```

To work properly, your application needs the deployment descriptor as given below.

The Deployment Descriptor

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<!-- Define servlet-mapped and path-mapped filters -->
<filter>
<filter-name>Trim Filter</filter-name>
<filter-class>TrimFilter</filter-class>
</filter>
<!-- Define filter mappings for the defined filters -->
<filter-mapping>
<filter-name>Trim Filter</filter-name>
<servlet-name>TrimFilteredServlet</servlet-name>
</filter-mapping>
<servlet>
<servlet-name>TrimFilteredServlet</servlet-name>
<servlet-class>TrimFilteredServlet</servlet-class>
</servlet>
</web-app>

```

When you run the servlet, it will first display something similar to the following.

The doGet method.



Notice that in this form the user enters a last name with a leading blank space. When the user submits the form, the browser will display the figure similar to the following.

The doPost method.



See how the trailing blank spaces have disappeared? This is evidence of the filter at work. Filtering the Response

You also can filter the response. In this example, you write a filter that appends a header and a footer of every servlet in the application. The filter code is given below.

The ResponseFilter Class

```
import java.io.*;  
import javax.servlet.Filter;  
import javax.servlet.FilterChain;  
import javax.servlet.FilterConfig;  
import javax.servlet.ServletContext;  
import javax.servlet.ServletException;  
import javax.servlet.ServletRequest;  
import javax.servlet.ServletResponse;  
import java.util.Enumeration;  
  
public class ResponseFilter implements Filter {  
    private FilterConfig filterConfig = null;  
    public void destroy() {  
        System.out.println("Filter destroyed");  
        this.filterConfig = null;  
    }  
  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        System.out.println("doFilter");
```

```

PrintWriter out = response.getWriter();
// this is added to the beginning of the PrintWriter
out.println("<HTML>");
out.println("<BODY>");
out.println("<CENTER>");
out.println("Page header");
out.println("<HR>");

chain.doFilter(request, response);
// this is added to the end of the PrintWriter
out.println("<HR>");
out.println("Page footer");
out.println("<CENTER>");
out.println("</BODY>");
out.println("</HTML>");
}

public void init(FilterConfig filterConfig) throws ServletException {
    System.out.println("Filter initialized");
    this.filterConfig = filterConfig;
}
}

```

An example of a servlet that is filtered is given below.

The ResponseFilteredServlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class ResponseFilteredServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<BR>Please enter your details.");
        out.println("<BR>");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<TABLE>");
        out.println("<TR>");
        out.println("<TD>First Name:</TD>");
        out.println("<TD><INPUT TYPE=TEXT NAME=firstName></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD>Last Name:</TD>");

```

```

out.println("<TD><INPUT TYPE=TEXT NAME=lastName></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>User Name:</TD>");
out.println("<TD><INPUT TYPE=TEXT NAME=userName></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>Password:</TD>");
out.println("<TD><INPUT TYPE=PASSWORD NAME=password></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("</TABLE>");
out.println("</FORM>");
}

}

```

For the example to work, you need a deployment descriptor, as given below.

The Deployment Descriptor

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- Define servlet-mapped and path-mapped filters -->
    <filter>
        <filter-name>Response Filter</filter-name>
        <filter-class>ResponseFilter</filter-class>
    </filter>

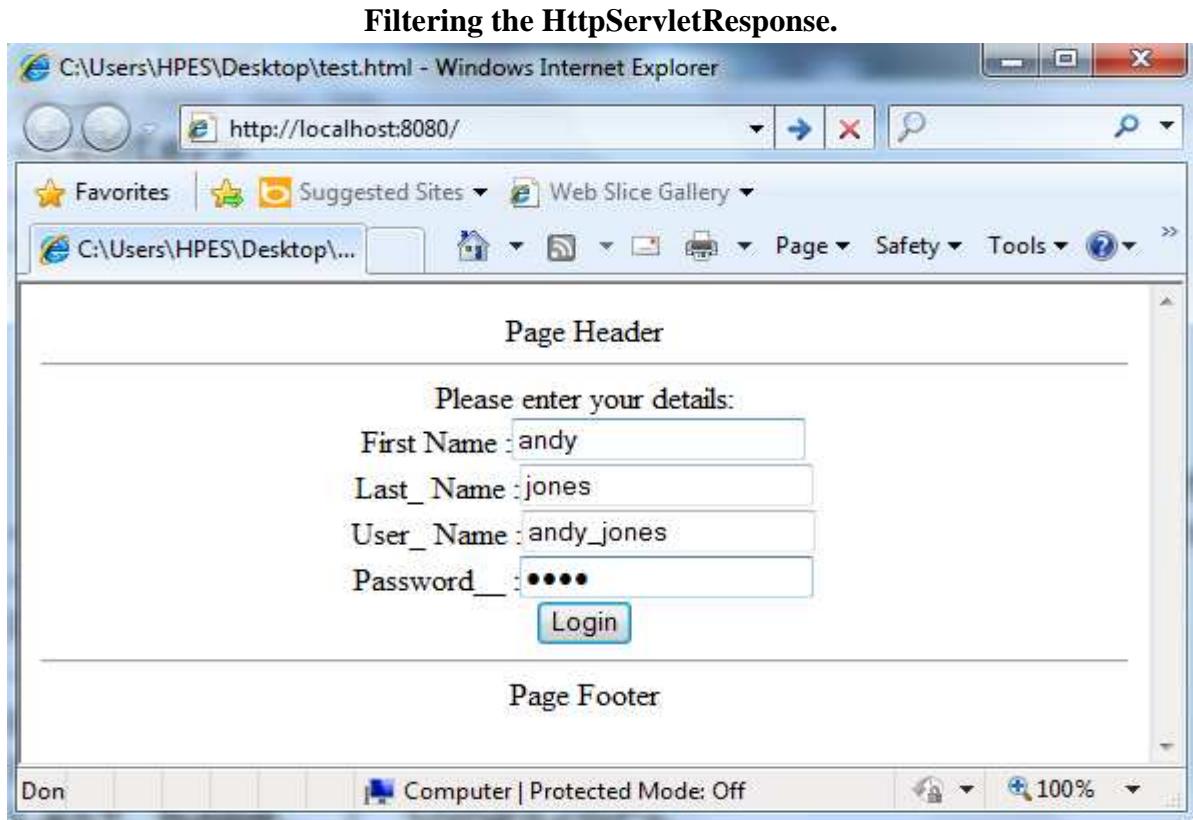
    <!-- Define filter mappings for the defined filters -->
    <filter-mapping>
        <filter-name>Response Filter</filter-name>
        <servlet-name>ResponseFilteredServlet</servlet-name>
    </filter-mapping>

    <servlet>
        <servlet-name>ResponseFilteredServlet</servlet-name>
        <servlet-class>ResponseFilteredServlet</servlet-class>
    </servlet>

```

```
</web-app>
```

The result of the previous filter is shown in the following figure.



When you view the source code for the HTML, it looks like this:

```
<HTML>
<BODY>
<CENTER>
Page header
<HR>
<BR>Please enter your details.
<BR>
<BR><FORM METHOD=POST>
<TABLE>
<TR>
<TD>First Name:</TD>
<TD><INPUT TYPE=TEXT NAME=firstName></TD> </TR>
```

```

<TR>
<TD>Last Name:</TD>
<TD><INPUT TYPE=TEXT NAME=lastName></TD> </TR>
<TR>
<TD>User Name:</TD>
<TD><INPUT TYPE=TEXT NAME=userName></TD> </TR>
<TR>
<TD>Password:</TD>
<TD><INPUT TYPE=PASSWORD NAME=password></TD> </TR>
<TR>
<TD ALIGN=RIGHT COLSPAN=2>
<INPUT TYPE=SUBMIT VALUE=Login></TD> </TR>
</TABLE>
</FORM>
<HR>

```

Page footer

```

<CENTER>
</BODY>
</HTML>

```

Filter Chain

You can apply more than one filter to a resource. In this example, you create the `UpperCaseFilter` and use the `TrimFilter` and a `DoublyFilteredServlet`. The examples are given below.

The `UpperCaseFilter`

```

import java.io.*;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import java.util.Enumeration;

public class UpperCaseFilter implements Filter {
    private FilterConfig filterConfig = null;
    public void destroy() {
        System.out.println("Filter destroyed");
        this.filterConfig = null;
    }
}

```

```

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
System.out.println("Filter");
Enumeration enum = request.getAttributeNames();
while (enum.hasMoreElements()) {
    String attributeName = (String) enum.nextElement();
    String attributeValue = (String)request.getAttribute(attributeName);
    request.setAttribute(attributeName, attributeValue.toUpperCase());    }
chain.doFilter(request, response);
}

public void init(FilterConfig filterConfig) throws ServletException {
    System.out.println("Filter initialized");
    this.filterConfig = filterConfig;
}
}

```

The DoublyFilteredServlet

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import com.brainysoftware.java.StringUtil;

public class DoublyFilteredServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>User Input Form</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<CENTER>");
        out.println("<BR>Please enter your details.");
        out.println("<BR>");
        out.println("<BR><FORM METHOD=POST>");
        out.println("<TABLE>");
        out.println("<TR>");
        out.println("<TD>First Name:</TD>");

```

```

out.println("<TD><INPUT TYPE=TEXT NAME=firstName></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>Last Name:</TD>");

out.println("<TD><INPUT TYPE=TEXT NAME=lastName></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>User Name:</TD>");
out.println("<TD><INPUT TYPE=TEXT NAME=userName></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>Password:</TD>");
out.println("<TD><INPUT TYPE=PASSWORD NAME=password></TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD ALIGN=RIGHT COLSPAN=2>");
out.println("<INPUT TYPE=SUBMIT VALUE=Login></TD>");
out.println("</TR>");
out.println("</TABLE>");
out.println("</FORM>");
out.println("</CENTER>");
out.println("</BODY>");
out.println("</HTML>");
}

```

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String firstName = (String) request.getAttribute("firstName");
String lastName = (String) request.getAttribute("lastName");
String userName = (String) request.getAttribute("userName");
String password = request.getParameter("password");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>Displaying Values</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("<CENTER>");
out.println("Here are your details.");

```

```

out.println("<TABLE>");
out.println("<TR>");
out.println("<TD>First Name:</TD>");
out.println("<TD>" + StringUtil.encodeHtmlTag(firstName) + "</TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>Last Name:</TD>");
out.println("<TD>" + StringUtil.encodeHtmlTag(lastName) + "</TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>User Name:</TD>");
out.println("<TD>" + StringUtil.encodeHtmlTag(userName) + "</TD>");
out.println("</TR>");
out.println("<TR>");
out.println("<TD>Password:</TD>");
out.println("<TD>" + StringUtil.encodeHtmlTag(password) + "</TD>");
out.println("</TABLE>");
out.println("</CENTER>");
out.println("</BODY>");
out.println("</HTML>");
}
}

```

Note:Notice that the filters in the filter chain modified attributes because you can't modify the parameters in the HTTP request object.

The deployment descriptor is given below.

The Deployment Descriptor for this Example

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<!-- Define servlet-mapped and path-mapped filters -->
<filter>
<filter-name>Trim Filter</filter-name>
<filter-class>TrimFilter</filter-class>
</filter>
<filter>
<filter-name>UpperCase Filter</filter-name>
<filter-class>UpperCaseFilter</filter-class>

```

```

</filter>
<!-- Define filter mappings for the defined filters -->
<filter-mapping>
    <filter-name>Trim Filter</filter-name>
    <servlet-name>DoublyFilteredServlet</servlet-name>
</filter-mapping>

<filter-mapping>
    <filter-name>UpperCase Filter</filter-name>
    <servlet-name>DoublyFilteredServlet</servlet-name>
</filter-mapping>

<servlet>
    <servlet-name>DoublyFilteredServlet</servlet-name>
    <servlet-class>DoublyFilteredServlet</servlet-class>
</servlet>
</web-app>

```

Finally, the following figure shows the two filters in action. User input is now trimmed and turned into uppercase characters.

The result of the doPost of the DoublyFilteredServlet.



JSP Basics

JavaServer Pages (JSP) is another Java technology for developing web applications. JSP was released during the time servlet technology had gained popularity as one of the best web technologies available. JSP is not meant to replace servlets, however. In fact, JSP is an extension of the servlet technology, and it is common practice to use both servlets and JSP pages in the same web applications.

Authoring JSP pages is so easy that you can write JSP applications without much knowledge of the underlying API. If you want to be a really good Java web programmer, however, you need to know both JSP and servlets. Even if you use only JSP pages in your Java web applications, understanding servlets is still very important. As you will see in this chapter and the chapters to come, JSP uses the same techniques as those found in servlet programming. For example, in JSP you work with HTTP requests and HTTP responses, request parameters, request attributes, session management, cookies, URL-rewriting, and so on. This chapter explains the relation between JSP and servlets, introduces the JSP technology, and presents many examples that you can run easily.

What's Wrong with Servlets?

The history of web server-side programming in Java started with servlets. Sun introduced servlets in 1996 as small Java-based applications for adding dynamic content to web applications. Not much later, with the increasing popularity of Java, servlets took off to become one of the most popular technologies for Internet development today.

Servlet programmers know how cumbersome it is to program with servlets, however, especially when you have to send a long HTML page that includes little code. Take the snippet given below as an example. The code is a fragment from a servlet-based application that displays all parameter names and values in an HTTP request.

Displays All Parameter/Value Pairs in a Request Using a Servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class MyDearServlet extends HttpServlet {
    //Process the HTTP GET request
    public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```

throws ServletException, IOException {
    doPost(request, response);
}

//Process the HTTP POST request
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Using Servlets</TITLE></HEAD>");
    out.println("<BODY BGCOLOR=#123123>");
    //Get parameter names
    Enumeration parameters = request.getParameterNames();
    String param = null;
    while (parameters.hasMoreElements()) {
        param = (String) parameters.nextElement();
        out.println(param + ":" + request.getParameter(param) + "<BR>");
    }
    out.println("</BODY>");
    out.println("</HTML>");
    out.close();
} //End of doPost method

/* other parts of the class goes here
.
.
.
*/
} //End of class

```

Nearly half of the content sent from the doPost method is static HTML. However, each HTML tag must be embedded in a String and sent using the println method of the PrintWriter object. It is a tedious chore. Worse still, the HTML page may be much longer.

Another disadvantage of using servlets is that every single change will require the intervention of the servlet programmer. Even a slight graphical modification, such as changing the value of the <BODY> tag's BGCOLOR attribute from #DADADA to #FFFFFF, will need to be done by the programmer (who in this case will work under the supervision of the more graphic-savvy web designer).

Sun understood this problem and soon developed a solution. The result was JSP technology. According to Sun's web site, "JSP technology is an extension of the servlet technology created to

support authoring of HTML and XML pages." Combining fixed or static template data with dynamic content is easier with JSP. Even if you're comfortable writing servlets, you will find in this chapter several compelling reasons to investigate JSP technology as a complement to your existing work.

What needs to be highlighted is that "JSP technology is an extension of the servlet technology." This means that JSP did not replace servlets as the technology for writing server-side Internet/intranet applications. In fact, JSP was built on the servlet foundation and needs the servlet technology to work.

JSP solves drawbacks in the servlet technology by allowing the programmer to intersperse code with static content, for example. If the programmer has to work with an HTML page template written by a web designer, the programmer can simply add code into the HTML page and save it as a .jsp file. If at a later stage the web designer needs to change the HTML body background color, he or she can do it without wasting the charging-by-the-hour programmer's time. He or she can just open the .jsp file and edit it accordingly.

The servlet code given above can be rewritten in JSP as shown below.

Displays All Parameter/Value Pairs in a Request Using JSP

```
<%@ page import="java.util.Enumeration" %> <HTML>
<HEAD><TITLE>Using JSP</TITLE></HEAD> <BODY BGCOLOR=#DADADA>
<%
//Get parameter names
Enumeration parameters = request.getParameterNames();  String param = null;
while (parameters.hasMoreElements()) {
    param = (String) parameters.nextElement();    out.println(param + ":" +
request.getParameter(param) +      "<BR>");
}
out.close();
%>
</BODY>
</HTML>
```

You can see that <HTML> tags stay as they are. When you need to add dynamic content, all you need to do is enclose your code in <% ... %> tags.

Again, JSP is not a replacement for servlets. Rather, JSP technology and servlets together provide an attractive solution to web scripting/programming by offering platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise, and most importantly, ease of use.

Running Your First JSP

This section invites you to write a simple JSP page and run it. The emphasis here is not on the architecture or syntax and semantics of a JSP page; instead the section demonstrates how to configure minimally the servlet/JSP container to run JSP. Tomcat 4 is used to run JSP applications. If you have installed and configured Tomcat 4 for your servlet applications, there is no more to do.

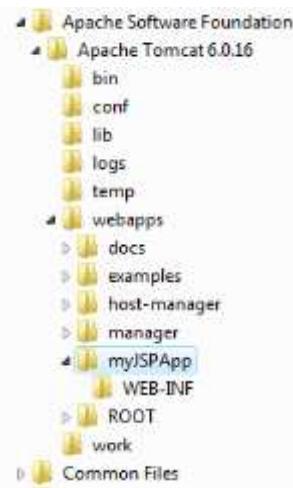
Note: In the JSP context, Tomcat is often referred to as a "JSP container." Because Tomcat also is used to run servlets, however, it is more common to call it a servlet/JSP container.

After reading this section, you will understand how much JSP simplifies things for servlets. To make your JSP page run, all you need to do is configure your JSP container (Tomcat) and write a JSP page. Configuration is only done once, at the beginning. No compilation is necessary.

Configuring Tomcat to Run a JSP Application

The first thing you need to do before you can run your JSP application is configure Tomcat so that it recognizes your JSP application. To configure Tomcat to run a particular JSP application, follow these steps:

1. Create a directory under %CATALINA_HOME%/webapps called myJSPApp. The directory structure is shown in the following figure.



2. Add a subdirectory named WEB-INF under the myJSPApp directory.
3. Edit server.xml, the server configuration file, so Tomcat knows about this new JSP application. The server.xml file is located in the conf directory under %CATALINA_HOME%. Open the file with your text editor and look for code similar to the following:

```
<Context path="/examples" docBase="examples" debug="0" reloadable="true">
.
.
.
</Context>
```

Right after the closing tag </Context>, add the following code:

```
<Context path="/myJSPApp" docBase="myJSPApp" debug="0" reloadable="true"> </Context>
```

4. Restart Tomcat.

Now you can write your JSP file and store it under the myJSPApp file. Alternatively, to make it more organized, you can create a subdirectory called jsp under myJSPApp and store your JSP files there. If you do this, you don't need to change the setting in the server.xml file.

Writing a JSP File

A JSP page consists of interwoven HTML tags and Java code. The HTML tags represent the presentation part and the code produces the contents. In its most basic form, a JSP page can include only the HTML part, like the code shown below.

The Simplest JSP Page

```
<HTML>
<HEAD>
</HEAD>
<BODY>
    JSP is easy.
</BODY>
</HTML>
```

Save this file as SimplePage.jsp in the myJSPApp directory. Now, start your web browser, and type the following URL: <http://localhost:8080/myJSPApp/SimplePage.jsp> The browser is shown in the following figure.



Other Examples

Of course, the code given above is not a really useful page, but it illustrates the point that a JSP page does not need to have code at all. If your page is purely static, like the one in above example, you shouldn't put it in a JSP file because JSP files are slower to process than HTML files. You

might want to use a JSP file for pure HTML tags, however, if you think the code might include Java code in the future. This saves you the trouble of changing all the links to this page at the later stage.

To write Java code in your JSP file, you embed the code in `<% ... %>` tags. For example, the following code is an example of intertwining Java code and HTML in a JSP file.

Interweaving HTML and Code

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<%
    out.println("JSP is easy");
%>
</BODY>
</HTML>
```

The code produced the same output. Notice, however, the use of the Java code to send the text. If you don't understand what `out.println` does, bear with me for a moment—it is discussed in detail in the next section. For now, knowing that `out.println` is used to send a String to the web browser is sufficient.

Notice also that the output of a JSP page is plain text consisting of HTML tags. No code section of the page will be sent to the browser.

Another example is given below. This snippet displays the string "Welcome. The server time is now" followed by the server time.

Displaying the Server Time

```
<HTML>
<HEAD>
    <TITLE>Displaying the server time</TITLE> </HEAD>
<BODY>
    Welcome. The server time is now
    <%
        java.util.Calendar now = java.util.Calendar.getInstance();  int hour =
        now.get(java.util.Calendar.HOUR_OF_DAY);  int minute =
        now.get(java.util.Calendar.MINUTE);
        if(hour<10)
            out.println("0" + hour);
        else
            out.println(hour);
        out.println(":");
    %>
```

```

if(minute<10)
    out.println("0" + minute);
else
    out.println(minute);
%>
</BODY>
</HTML>

```

The code in the above example displays the time in the hh:mm format. Therefore, if the hour is less than 10, a "0" precedes, which means that nine will be displayed as 09 instead of 9.

How JSP Works

Inside the JSP container is a special servlet called the page compiler. The servlet container is configured to forward to this page compiler all HTTP requests with URLs that match the .jsp file extension. This page compiler turns a servlet container into a JSP container. When a .jsp page is first called, the page compiler parses and compiles the .jsp page into a servlet class. If the compilation is successful, the .jsp servlet class is loaded into memory. On subsequent calls, the servlet class for that .jsp page is already in memory; however, it could have been updated. Therefore, the page compiler servlet will always compare the timestamp of the .jsp servlet with the .jsp page. If the .jsp page is more current, recompilation is necessary. With this process, once deployed, JSP pages only go through the time-consuming compilation process once.

You may be thinking that after the deployment, the first user requests for a .jsp page will experience unusually slow response due to the time spent for compiling the .jsp file into a .jsp servlet. To avoid this unpleasant situation, a mechanism in JSP allows the .jsp pages to be pre-compiled before any user request for them is received. Alternatively, you deploy your JSP application as a web archive file in the form of a compiled servlet. This technique is called Application Deployment."

The JSP Servlet Generated Code

When the JSP is invoked, Tomcat creates two files in the C:\%CATALINA_HOME%\work\localhost\examples\jsp directory. Those two files are SimplePage_jsp.java and SimplePage_jsp.class. When you open the SimplePage_jsp.java file, you will see the following:

```

package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.jasper.runtime.*;
public class SimplePage_jsp extends HttpJspBase {

```

```

static {

}

public SimplePage_jsp( ) {

}

private static boolean _jspx_initited = false;

public final void _jspx_init() throws org.apache.jasper.JasperException { }

public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
String _value = null;
try {
if (_jspx_initited == false) {
synchronized (this) {
if (_jspx_initited == false) {
_jspx_init();
_jspx_initited = true;
}
}
}
_jspxFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html;charset=ISO-8859-1");
pageContext = jjspxFactory.getPageContext(this,
request, response, "", true, 8192, true);
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
// begin
[file="C:\\tomcat4\\bin\\..\\webapps\\examples\\jsp\\SimplePage.jsp";from=(0, 2);to=(2,0)]
out.println("JSP is easy");
// end // HTML // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\examples\\jsp\\SimplePage.jsp";from=(2, 2);to=(3,0)]
out.write("\r\n");
}

```

```

// end
}
catch (Throwable t) {
    if (out != null && out.getBufferSize() != 0)
        out.clearBuffer();
    if (pageContext != null)
        pageContext.handlePageException(t);
}
finally {
    if (_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);      }
}
}

```

For now, A full explanation of the preceding code is deferred until you learn more about how the interfaces and classes are used to run a JSP page. You can read the section, "The Generated Servlet Revisited," later in this chapter, to learn more about the code listed here.

The JSP API

The JSP technology is based on the JSP API that consists of two packages: javax.servlet.jsp and javax.servlet.jsp.tagext. This chapter will discuss the classes and interfaces of the javax.servlet.jsp package and javax.servlet.jsp.tagext will be discussed later. In addition to these two packages, JSP also needs the two servlet packages—javax.servlet and javax.servlet.http. When you study the javax.servlet.jsp package, you will know why we say that JSP is an extension of servlet technology and understand why it is important that a JSP application programmer understands the servlet technology well. The javax.servlet.jsp package has two interfaces and four classes. The interfaces are as follows:

- *JspPage*
- *HttpJspPage*

The four classes are as follows:

- *JspEngineInfo*
- *JspFactory*
- *JspWriter*
- *PageContext*

In addition, there are also two exception classes: *JspException* and *JspError*.

The JspPage Interface

The JspPage is the interface that must be implemented by all JSP servlet classes. This may remind you of the javax.servlet.Servlet interface, of course. And, not surprisingly, the JspPage interface does extend the javax.servlet.Servlet interface. The JSPPage interface has two methods, JspInit and JspDestroy, whose signatures are as follows:

```
public void jspInit()  
public void jspDestroy()
```

jspInit, which is similar to the init method in the javax.servlet.Servlet interface, is called when the JspPage object is created and can be used to run some initialization. This method is called only once during the life cycle of the JSP page: the first time the JSP page is invoked. The jspDestroy method is analogous with the destroy method of the javax.servlet.Servlet interface. This method is called before the JSP servlet object is destroyed. You can use this method to do some clean-up, if you want.

Most of the time, however, JSP authors rarely make full use of these two methods. The following example illustrates how you can implement these two methods in your JSP page: <%!

```
public void jspInit() {  
    System.out.println("Init");  
}  
public void jspDestroy() {  
    System.out.println("Destroy");  
}  
>  
<%  
    out.println("JSP is easy");  
>
```

Notice that the first line of the code starts with <%!. You will find the explanation of this construct in the next chapter, "JSP Syntax."

The HttpJspPage Interface

This interface directly extends the JspPage interface. There is only one method: _jspService. This method is called by the JSP container to generate the content of the JSP page. The _jspService has the following signature:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException.
```

You can't include this method in a JSP page, such as in the following code:

```
<%!  
public void jspInit() {  
    System.out.println("Init");  
}
```

```

public void jspDestroy() {
    System.out.println("Destroy");
}
public void _jspService(HttpServletRequest request,    HttpServletResponse response)
throws ServletException, IOException {
    System.out.println("Service");
}
%>

```

The JspFactory Class

The JspFactory class is an abstract class that provides methods for obtaining other objects needed for the JSP page processing. The class has the static method getDefaultFactory that returns a JspFactory object. From the JspFactory object, a PageContext and a JspEngineInfo object can be obtained that are useful for the JSP page processing. These objects are obtained using the JspFactory class's getEngineInfo method and the getPageContext method, whose signatures are given here:

```

public abstract JspEngineInfo getEngineInfo()
public abstract PageContext getPageContext ( Servlet requestingServlet, ServletRequest request,
ServletResponse response, String errorPageURL, boolean needsSession, int buffer, boolean
autoFlush)

```

The following code is part of the _jspService method that is generated by the JSP container:

```

JspFactory _jspxFactory = null;
PageContext pageContext = null;
_jspxFactory = JspFactory.getDefaultFactory(); .

```

```
pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 8192, true);
```

The JspEngineInfo Class

The JspEngineInfo class is an abstract class that provides information on the JSP container. Only one method, getSpecificationVersion, returns the JSP container's version number. Because this is the only method currently available, this class does not have much use.

You can obtain a JspEngineInfo object using the getEngineInfo method of the JspFactory class.

The PageContext Class

PageContext represents a class that provides methods that are implementation-dependent. The PageContext class itself is abstract, so in the _jspService method of a JSP servlet class, a PageContext object is obtained by calling the getPageContext method of the JspFactory class.

The PageContext class provides methods that are used to create other objects. For example, its getOut method returns a JspWriter object that is used to send strings to the web browser. Other methods that return servlet-related objects include the following:

- getRequest, returns a ServletRequest object
- getResponse, returns a ServletResponse object
- getServletConfig, returns a ServletConfig object
- getServletContext, returns a ServletContext object
- getSession, returns an HttpSession object

The JspWriter Class

The JspWriter class is derived from the java.io.Writer class and represents a Writer that you can use to write to the client browser. Of its many methods, the most important are the print and println methods. Both provide enough overloads that ensure you can write any type of data. The difference between print and println is that println always adds the new line character to the printed data.

Additional methods allow you to manipulate the buffer. For instance, the clear method clears the buffer. It throws an exception if some of the buffer's content has already been flushed. Similar to clear is the clearBuffer method, which clears the buffer but never throws any exception if any of the buffer's contents have been flushed.

The Generated Servlet Revisited

Now that you understand the various interfaces and classes in the javax.servlet.jsp package, analyzing the generated JSP servlet will make more sense. When you study the generated code, the first thing you'll see is the following:

```
public class SimplePage_jsp extends HttpJspBase {
```

The discussion should start with the questions, "What is HttpJspBase?" and "Why doesn't the JSP servlet class implement the javax.servlet.jsp.JspPage or javax.servlet.jsp.HttpJspPage interface?"

First, remember that the JSP specification defines only standards for writing JSP pages. A JSP page itself will be translated into a java file, which in turn will be compiled into a servlet class. The two processes are implementation dependent and do not affect the way a JSP page author codes. Therefore, a JSP container has the freedom and flexibility to do the page translation in its own way. The JSP servlet-generated code presented in this chapter is taken from Tomcat. You can therefore expect a different Java file to be generated by other JSP containers.

In Tomcat, HttpJspBase is a class whose source can be found under the
src\jasper\src\share\org\apache\jasper\runtime directory under the %CATALINA_HOME%
directory.

Most importantly, the signature of this class is as follows:

```
public abstract class HttpJspBase extends HttpServlet implements HttpJspPage
```

Now you can see that HttpJspBase is an HttpServlet and it does implement the
javax.servlet.jsp.HttpJspPage interface. The HttpJspBase is more like a wrapper class so that its

derived class does not have to provide implementation for the interface's methods if the JSP page author does not override them. The complete listing of the class follows:

```
package org.apache.jasper.runtime;
```

```
import java.io.IOException;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.MalformedURLException;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.JasperException;
import org.apache.jasper.Constants;

public abstract class HttpJspBase extends HttpServlet implements HttpJspPage {

    protected PageContext pageContext;
    protected HttpJspBase() {
    }

    public final void init(ServletConfig config)
        throws ServletException {
        super.init(config);
        jspInit();
    }

    public String getServletInfo() {
        return Constants.getString ("jsp.engine.info");
    }

    public final void destroy() {
        jspDestroy();
    }

    public final void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        _jspService(request, response);
    }

    public void jspInit() {
    }
```

```

public void jspDestroy() {
}
public abstract void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException;
}

```

Back to the generated JSP servlet class, remember that the JSP page source code is simply the following three lines of code:

```

<%
    out.println("JSP is easy");
%>

```

There are no `jspInit` and `jspDestroy` methods in the source code, so there are no implementations for these two methods in the resulting servlet code. The three lines of code, however, translate into the `_jspService` method. For reading convenience, the method is reprinted here. Notice that, for clarity, the comments have been removed.

```

public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws java.io.IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    try {
        if (_jspx_initited == false) {
            synchronized (this) {
                if (_jspx_initited == false) {
                    _jspx_init();
                    _jspx_initited = true;
                }
            }
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=ISO-8859-1");
        pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 8192, true);
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
    }
}

```

```

out.println("JSP is easy");
out.write("\r\n");
}
catch (Throwable t) {
    if(out != null && out.getBufferSize() != 0)
        out.clearBuffer();
    if(pageContext != null)
        pageContext.handlePageException(t);
}
finally {
    if(_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);
}
}

```

Implicit Objects

Having examined the generated JSP servlet source code, you know that the code contains several object declarations in its `_jspService` method. Recall this part from the code in the preceding section:

```

public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
String _value = null;
try {
    .
    .
    .
_jspxFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html;charset=ISO-8859-1");
pageContext = jjspxFactory.getPageContext(this, request, response, "", true, 8192, true);
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
.
.
.
}

```

You see that there are object references, such as pageContext, session, application, config, out, and so on. These object references are created whether they are used from inside the page. They are automatically available for the JSP page author to use! These objects are called implicit objects and are summarized in the following table.

JSP Implicit Objects

<u>Object</u>	<u>Type</u>
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext
page	javax.servlet.jsp.HttpJspPage
exception	java.lang.Throwable

By looking at above table, you now know why you can send a String of text by simply writing:

```
<%
    out.println("JSP is easy.");
%>
```

In the case of the code above, you use the implicit object out that represents a javax.servlet.jsp.JspWriter. All the implicit objects are discussed briefly in the following subsections.

request and response Implicit Objects

In servlets, both objects are passed in by the servlet container to the service method of the javax.servlet.http.HttpServlet class. Remember its signature?

```
protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

In a servlet, before you send any output, you are required to call the setContentType of the HttpServletResponse to tell the browser the type of the content, as in the following code:

```
response.setContentType("text/html");
```

In JSP, this is done automatically for you in the generated JSP servlet class, as follows:

```
response.setContentType("text/html;charset=ISO-8859-1");
```

Having an HttpServletRequest and an HttpServletResponse, you can do anything you like as you would in a servlet. For example, the following JSP page retrieves the value of a parameter called firstName and displays it in the browser:

```
<HTML>
<HEAD>
<TITLE>Simple Page</TITLE>
</HEAD>
<BODY>
<%
    String firstName = request.getParameter("firstName");
    out.println("First name: " + firstName);
%>
</BODY>
</HTML>
```

The following example uses the sendRedirect method of the javax.servlet.http.HttpServletResponse to redirect the user to a different URL: <% response.sendRedirect("http://www.newriders.com"); %>

out Implicit Object

out is probably the most frequently used implicit object. You call either its print method or its println method to send text or other data to the client browser. In a servlet, you always need to call the getWriter method of the javax.servlet.http.HttpServletResponse interface to obtain a PrintWriter before you can output anything to the browser, as follows:

```
PrintWriter out = response.getWriter();
```

In JSP, you don't need to do this because you already have an out that represents a javax.servlet.jsp.JspWriter object.

session Implicit Object

The session implicit object represents the HttpSession object that you can retrieve in a servlet by calling the getSession method of the javax.servlet.http.HttpServletRequest interface, as in the following code:

```
request.getSession();
```

The following code is a JSP page that uses a session object to implement a counter:

```
<HTML>
<HEAD>
<TITLE>Counter</TITLE>
</HEAD>
<BODY>
<%
    
```

```

String counterAttribute = (String) session.getAttribute("counter");
int count = 0;
try {
    count = Integer.parseInt(counterAttribute);
}
catch (Exception e) {
}
count++;
session.setAttribute("counter", Integer.toString(count));
out.println("This is the " + count + "th time you visited this page in this session.");
%>
</BODY>
</HTML>

```

See how session is readily available without efforts from the programmer's side?

Warning! Note that the session object is available only in a JSP page that participates in the session management.

application

The application implicit object represents the javax.servlet.ServletContext object. In an HttpServlet, you can retrieve the ServletContext method by using the getServletContext method.

config

The config implicit object represents a javax.servlet.ServletConfig object that in a servlet can be retrieved by using the getServletConfig method.

pageContext

The pageContext implicit object represents the javax.servlet.jsp.PageContext object.

page

The page implicit object represents the javax.servlet.jsp.HttpJspPage interface.

exception

The exception object is available only on pages that have been defined as error pages.

Advanced JSP

In the previous chapter, "JSP Basics," you learned that a JSP page can have Java code and HTML tags. More formally, you can say that a JSP page has elements and template data. The elements, which also are called JSP tags, make up the syntax and semantics of JSP. Template data is everything else. Template data includes parts that the JSP container does not understand, such as HTML tags.

There are three types of elements:

- *Directive elements*
- *Scripting elements*
- *Action elements*

To write an effective JSP page, you need to understand all these elements well. Elements have two forms: the XML form and the `<% ... %>` alternative form. The conversion between the XML form and the alternative form is presented at the end of the chapter. Template data remains as it is, normally passed through the client uninterpreted.

This chapter discusses the three types of JSP elements and comments. It also presents examples on how to use these elements. You will also learn how incorporating these elements affects the JSP servlets—servlets that result from the translation of JSP pages.

Directives

Directives are messages to the JSP container containing information on how the JSP container must translate a JSP page into a corresponding servlet. Directives have the following syntax:

```
<%@ directive (attribute="value")* %>
```

The asterisk (*) means that what is enclosed in the brackets can be repeated zero or more times. The syntax can be re-written in a more informal way as follows:

```
<%@ directive attribute1="value1" attribute2="value2" ... %>
```

White spaces after the opening `<% @` and before the closing `%>` are optional, but are recommended to enhance readability.

Warning! Note that an attribute value must be quoted.

The three types of directives are as follows:

- Page directives
- Include directives
- Tag library directives

The first two directives are discussed in detail in this chapter. Discussion on the tag library directive is deferred until the chapter "Using JSP Custom Tags."

The Page Directive

The Page directive has the following syntax:

```
<%@ page (attribute="value")* %>
```

Or, if you want to use the more informal syntax:

```
<%@ page attribute1="value1" attribute2="value2" ... %>
```

The Page directive supports 11 attributes. These attributes are summarized in the following table.

The Page Directive's Attributes

<u>Attribute</u>	<u>Value Type</u>	<u>Default Value</u>
language	Scripting language name	"java"
Info	String	Depends on the JSP container
contentType	MIME type, character set	"text/html;charset=ISO-8859-1"
extends	Class name	None
import	Fully qualified class name or package name	None
buffer	Buffer size or false	8192
autoFlush	Boolean	"true"
session	Boolean	"true"
isThreadSafe	Boolean	"true"
errorPage	URL	None
isErrorHandler	Boolean	"false"

An example of the use of the Page directive is as follows: `<%@ page buffer="16384" session="false" %>` With JSP, you can specify multiple page directives in your JSP page, such as the following:

```
<%@ page buffer="16384" %>
<%@ page session="false" %>
```

Except for the import attribute, however, JSP does not allow you to repeat the same attribute within a page directive or in multiple directives on the same page.

The following is illegal because the info attribute appears more than once in a single page:

```
<%@ page info="Example Page" %>
<%@ page buffer="16384" %>
<%@ page info="Unrestricted Access" %>
```

The following also is illegal because the buffer attribute appears more than once in the same page directive:

```
<%@ page buffer="16384" info="Example Page" buffer="8192" %>
```

The following is legal, however, because the import attribute can appear multiple times:

```
<%@ page import="java.io.*" info="Example Page" %>
<%@ page import="java.util.Enumeration" %>
<%@ page import="com.brainysoftware.web.FileUpload" %>
```

Alternatively, imported libraries can appear in a single import attribute, separated by commas, as shown in the following code:

```
<%@ page import="java.io.*, java.util.Enumeration" %>
```

The following section discusses each attribute in more detail and examines the effect on the generated servlet.

The language Attribute

The language attribute specifies the scripting language used in the JSP page. By default, the value is "java" and all JSP containers must support Java as the scripting language. With Tomcat, this is the only accepted language. Other JSP containers might accept different languages, however.

Specifying this attribute, as in the following code, does not have any effect on the generated servlet:

```
<%@ page language="java" %>
<%
    out.println("JSP is easy");
%>
```

In fact, this attribute is useful only in a JSP container that supports a language other than Java as the scripting language.

The info Attribute

The info attribute allows you to insert any string that later can be retrieved using the getServletInfo method. For example, you can add the following page directive with an info attribute:

```
<%@ page info="Written by Bulbul" %>
```

The JSP container then will create a public method getServletInfo in the resulting servlet. This method returns the value of the info attribute specified in the page. For the previous page directive, the getServletInfo method will be written as follows:

```
public String getServletInfo() {
    return "Written by Bulbul";
}
```

In the same JSP page, you can retrieve the info attribute's value by calling the getServletInfo method. For instance, the following JSP page will return "Written by Bulbul" on the client browser.

```
<%@ page info="Written by Bulbul" %>
<%
    out.println(getServletInfo());
%>
```

The default value for this attribute depends on the JSP container.

The contentType Attribute

The contentType attribute defines the Multipurpose Internet Mail Extension (MIME) type of the HTTP response sent to the client. The default value is "text/html;charset=ISO-8859-1" and this is reflected in _jspService method in the generated servlet. For example, this is the cleaned up version of the servlet that is generated from a JSP page that does not specify the contentType attribute in its page directive; that is, the default value is used:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException { . . .

try {
    . . .
    . . .

_jspFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html;charset=ISO-8859-1");
    . . .
    . . .

}
catch (Throwable t) {
    . . .
    . . .

}
finally {
    . . .
    . . .

} }
```

You will want to use this attribute when working with pages that need to send characters in other encoding schemes. For example, the following directive tells the JSP container that the output should be sent using simplified Chinese characters:

```
<%@ page contentType="text/html;charset=GB2312" %>
```

The extends Attribute

The extends attribute defines the parent class that will be inherited by the generated servlet. You should use this attribute with extra care. In most cases, you should not use this attribute at all. In Tomcat, the parent class that will be subclassed by the resulting servlet is HttpJspBase.

The import Attribute

The import attribute is similar to the import keyword in a Java class or interface. The attribute is used to import a class or an interface or all members of a package. You will definitely use this attribute often. Whatever you specify in the import attribute of a page directive will be translated into an import statement in the generated servlet class. By default, Tomcat specifies the following import statements in every generated servlet class. You don't need to import what has been imported by default:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.jasper.runtime.*;
```

As an example, consider the following JSP page that imports the java.io package and the java.util.Enumeration interface:

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.Enumeration" %>
```

The two will be added before the default import statements in the generated servlet class, as described in the following code fragment:

```
import java.io.*;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.jasper.runtime.*;
```

The buffer Attribute

By default, a JSP page's content is buffered to increase performance. The default size of the buffer is 8Kb or 8192 characters.

Consider an example that specifies the buffer attribute with a size of 16Kb:

```
<%@ page buffer="16kb" %>
```

This attribute is reflected in the `_jspService` method in the generated servlet class, as in the following cleaned up code. The buffer size is passed to the `getPageContext` method of the `javax.servlet.jsp.JspFactory` class when creating a `PageContext` object.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException { .
.
.
try {
.
.
.
_jspFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html;charset=ISO-8859-1");
// 8192 in the following line represents the buffer size
pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 16384, true);
.
.
.
}
catch (Throwable t) {
.
.
.
}
finally {
.
.
.
}
}
```

With the buffer attribute in a page directive, you can do two things:

- Eliminate the buffer by specifying the value "none", as in the following code: `<% @ page buffer="none" %>`

If you decide not to use a buffer, the `getPageContext` method will be invoked passing 0 as the buffer size, as in the following code:

```
pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 0, true);
```

- Change the size of the buffer by assigning a number to the attribute. The attribute value represents the number in kilobytes. Therefore, "16" means 16Kb alias 16384 characters. You can either write the number only, or the number plus "Kb". For example, "16" is the same as "16Kb".

The following example shows two page directives in two different JSP pages. The first cancels the use of the buffer, and the second changes the size of the buffer to 12Kb.

```
<%@ page import="java.io.*" buffer="none" %>
<%@ page import="java.io.*" buffer="12" %>
```

Warning! Note that the JSP container has the discretion to use a buffer size larger than specified to improve performance.

The autoFlush Attribute

The autoFlush attribute is related to the page buffer. When the value is "true", the JSP container will automatically flush the buffer when the buffer is full. If the value is "false", however, the JSP author needs to flush the buffer manually using the flush method of the JspWriter object, such as the following:

```
out.flush();
```

For example, the following code specifies a false value for the autoFlush attribute:

```
<%@ page autoFlush="false" %>
```

The session Attribute

By default, a JSP page participates in the JSP container's session management. This is indicated by the declaration of a javax.servlet.http.HttpSession object reference and the creation of it through the getSession method of the javax.servlet.jsp.PageContext, as illustrated by the following code:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
// declare an HttpSession object reference
HttpSession session = null;
try {
.
.
.
_jspxFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html;charset=ISO-8859-1");
pageContext = _jspxFactory.getPageContext( this, request, response, "", true, 0, true);
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
```

```

// create a HttpSession object
session = pageContext.getSession();
.
.
.
}
catch (Throwable t) {
.
.
.
}
finally {
.
.
.
}
}

```

Whether your JSP page participates in the session management is determined by the value of the session attribute in a page directive. By default this value is true; that is, the JSP page is part of the session management. I want to make an important point, however: You should not let your JSP page participate in the session management unnecessarily because this consumes resources. Because of this, you should always use the session attribute and assign it the value of "false" unless you are specifically needing session management.

When the value of this attribute is "false", no javax.servlet.http.HttpSession object reference is declared and no HttpSession object is created. As a result, the session implicit object is not available in the JSP page.

The following shows an example of the use of the session attribute:

```
<%@ page session="false" %>
```

The isThreadSafe Attribute

As discussed in the chapter , "Inside Servlets," you can make a servlet thread-safe by inheriting the javax.servlet.SingleThreadModel. You can control this behavior in the JSP page by using the isThreadSafe attribute, which by default has the value of "true". When the value of this attribute is true, you guarantee that simultaneous access to this page is safe. However, you can assign "false" to this attribute as in the following code:

```
<%@ page isThreadSafe="false" %>
```

By setting the isThreadSafe attribute to false, you are telling the JSP translator that you (the programmer of this page) cannot guarantee that simultaneous accesses to this page will be safe. Therefore, you are asking the JSP translator to make this page thread-safe. This will make the

generated servlet implement the javax.servlet.SingleThreadModel interface, as in the following signature of the JSP servlet class:

```
public class SimplePage_jsp extends HttpJspBase implements SingleThreadModel
```

In other words, when the value of this attribute is false, the JSP container will serialize multiple requests for this page; that is, the JSP container will wait until the JSP servlet finishes responding to a request before passing another request to it.

The `errorCode` Attribute

The `errorCode` attribute specifies the URL of an error page that will be displayed if an uncaught exception occurs in this current JSP page. Referring to the URL of the error page is sometimes tricky. The easiest solution is to store the error page in the same directory as the current JSP page. In this case, you need only to mention the name of the error page, as in the following example:

```
<%@ page errorCode="ErrorPage.jsp" %>
<%
    String name = request.getParameter("otherName");
    // this will throw an exception because the parameter "otherName"
    // does not exist, so name will be null.
    // this will cause the ErrorPage.jsp to be displayed
    name.substring(1, 1);
%>
```

When the page is called, the error page is displayed. An error page must specify the `isErrorPage` attribute in its page directive and the value of this attribute must be "true". The following is an `errorCode` attribute that is assigned an error page in a URL:

```
<%@ page errorCode="/myJspApp/ErrorPage.jsp" %>
```

The `isErrorPage` Attribute

The `isErrorPage` attribute can accept the value of "true" or "false", and the default value is "false". It indicates whether the current JSP page is an error page; that is, the page that will be displayed when an uncaught exception occurs in the other JSP page. If the current page is an error page, it has access to the exception implicit object.

The `include` Directive

The `include` directive is the second type of the JSP directive elements. This directive enables JSP page authors to include the contents of other files in the current JSP page.

The `include` directive is useful if you have a common source that will be used by more than one JSP page. Instead of repeating the same code in every JSP page, thus creating a maintenance problem, you can place the common code in a separate file and use an `include` directive from each JSP page.

The included page itself can be static, such as an HTML file, or dynamic, such as another JSP page. If you are including a JSP page, the included JSP page itself can include another file. Therefore, nesting include directives is permitted in JSP. The syntax for the include directive is as follows:

```
<%@ include file="relativeURL" %>
```

The following is an example of how to include HTML files:

```
<%@ include file="includes/header.html" %> <%
out.println("<BODY>");
// other content;
%>
<%@ include file="includes/footer.html" %>
```

Note: If the relativeURL part begins with a forward slash character (/), it is interpreted as an absolute path on the server. If relativeURL does not start with a "/", it is interpreted as relative to the current JSP page.

Now, we'll look at how included files are translated into the JSP servlet file. The following example is a simple JSP page that includes two HTML files: Header.html and Footer.html. Both included files are located under the includes subdirectory, which itself is under the directory that hosts the current JSP file. The JSP page is called SimplePage.jsp and is given below. The page simply displays the current server time. What you are interested in here is the generated servlet code.

A Simple JSP Page that Includes Two Files

```
<%@ page session="false" %>
<%@ page import="java.util.Calendar" %>
<%@ include file="includes/Header.html" %>
<% out.println("Current time: " + Calendar.getInstance().getTime()); %>
<%@ include file="includes/Footer.html" %>
```

The Header.html File

```
<HTML>
<HEAD>
<TITLE>Welcome</TITLE>
<BODY>
```

The Footer.html File

```
</BODY>
</HTML>
```

The generated servlet is as follows:

```
package org.apache.jsp;
import java.util.Calendar;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.jasper.runtime.*;

public class SimplePage_jsp extends HttpJspBase {
    static {
    }
    public SimplePage_jsp( ) {
    }
    private static boolean _jspx_initited = false;
    public final void _jspx_init() throws org.apache.jasper.JasperException { }
    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {
            if (_jspx_initited == false) {
                synchronized (this) {
                    if (_jspx_initited == false) {
                        _jspx_init();
                        _jspx_initited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request, response, "", false, 8192, true);
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            out = pageContext.getOut();
            // HTML // begin
        }
    }
}
```

```

[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(0,27); to=(1,0)]
    out.write("\\r\\n");
    // end    // HTML // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(1,39); to=(2,0)]
    out.write("\\r\\n");
    // end
    // HTML    // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\includes\\Header.html"; from=(0,0);to=(4,0)]
    out.write("<HTML>\\r\\n<HEAD>\\r\\n<TITLE>Welcome</TITLE>\\r\\n<BODY>\\r\\n");
    // end
    // HTML // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(2,42); to=(3,0)]
    out.write("\\r\\n");
    // end
    // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(3,2);to =(5,0)]
    out.println("Current time: " + Calendar.getInstance().getTime());
    // end
    // HTML // begin
    [file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(5,2);to =(6,0)]
    out.write("\\r\\n");
    // end
    // HTML // begin
[file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\includes\\Footer.html"; from=(0,0);to=(2,0)]
    out.write("</BODY>\\r\\n</HTML>\\r\\n");
    // end
    // HTML    // begin
    [file="C:\\tomcat4\\bin\\..\\webapps\\myJSPApp\\SimplePage.jsp";from=(6,42); to=(7,0)]
    out.write("\\r\\n");
    // end
}
catch (Throwable t) {
    if (out != null && out.getBufferSize() != 0)      out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
    finally {
        if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
    }
}

```

The lines in bold denote the beginning of the included file. Because all the included files are placed inline before being compiled, including files does not have any performance effect on the application.

The taglib Directive

The taglib, or tag library, directive can be used to extend JSP functionality. This is a broad topic and has been given a chapter of its own. Have a little patience.

Scripting Elements

Scripting elements allow you to insert Java code in your JSP pages. There are three types of scripting elements:

- Scriptlets
- Declarations
- Expressions

The three elements are discussed in the following sections.

Scriptlets

Throughout the previous chapters and up to this point in this chapter, you have seen scriptlets in the examples. Scriptlets are the code blocks of a JSP page. Scriptlets start with an opening `<%` tag and end with a closing `%>` tag.

Note: Directives also start with `<%` and end with `%>`. However, in a directive a `@` follows the `<%`.

The following JSP page is an example of using scriptlets. In the JSP page, you try to connect to a database and retrieve all records from a table called Users. Among the fields in the Users table are FirstName, LastName, UserName, and Password. Upon obtaining a ResultSet object, you display all the records in an HTML table. The JSP page is given below.

Displaying Database Records

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    System.out.println("JDBC driver loaded");
}
catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}
%>
<HTML>
<HEAD>
<TITLE>Display All Users</TITLE>
</HEAD>
<BODY>
```

```

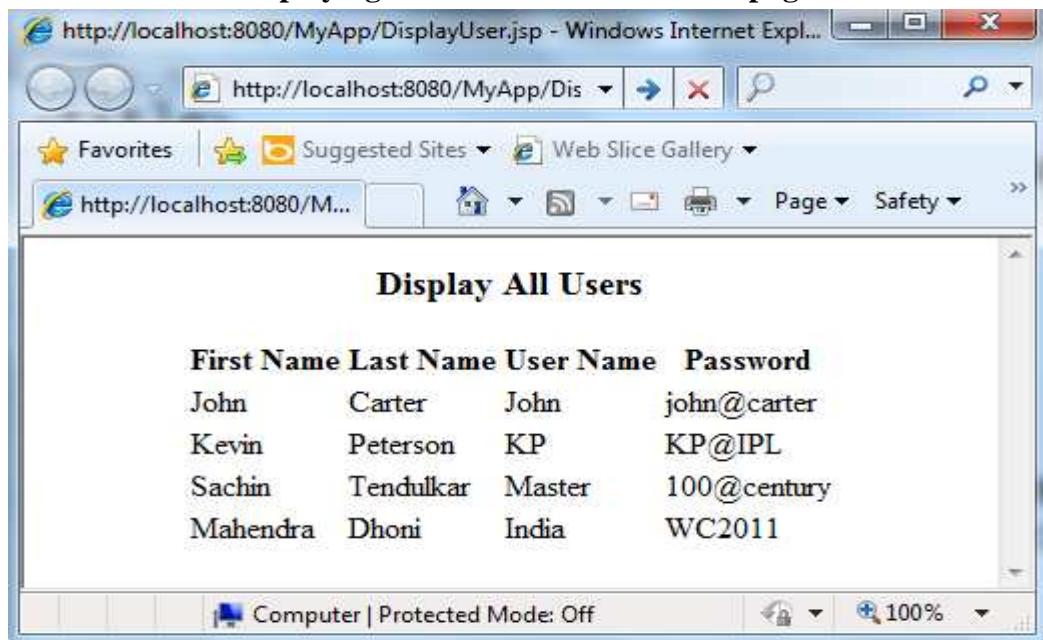
<CENTER>
<BR><H2>Displaying All Users</H2>
<BR>
<BR>
<TABLE>
<TR>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>User Name</TH>
<TH>Password</TH>
</TR>
<%
String sql = "SELECT FirstName, LastName, UserName, Password" + " FROM Users";
try {
    Connection con = DriverManager.getConnection("jdbc:odbc:JavaWeb");
    Statement s = con.createStatement();
    ResultSet rs = s.executeQuery(sql);

    while (rs.next()) {
        out.println("<TR>");
        out.println("<TD>" + rs.getString(1) + "</TD>");
        out.println("<TD>" + rs.getString(2) + "</TD>");
        out.println("<TD>" + rs.getString(3) + "</TD>");
        out.println("<TD>" + rs.getString(4) + "</TD>");
        out.println("</TR>");
    }
    rs.close();
    s.close();
    con.close();
}
catch (SQLException e) {
}
catch (Exception e) {
}
%>
</TABLE>
</CENTER>
</BODY>
</HTML>

```

The result of this JSP page in a browser is given below.

Displaying database records in a JSP page.



A screenshot of a Windows Internet Explorer window. The title bar says "http://localhost:8080/MyApp/DisplayUser.jsp - Windows Internet Expl...". The main content area has a heading "Display All Users" and a table with four columns: First Name, Last Name, User Name, and Password. The data is as follows:

First Name	Last Name	User Name	Password
John	Carter	John	john@carter
Kevin	Peterson	KP	KP@IPL
Sachin	Tendulkar	Master	100@century
Mahendra	Dhoni	India	WC2011

Note that you don't do any HTML encoding to the data returned by the `getString` method of the `ResultSet` because you don't know yet how to write and use a method in a JSP page. If, for example, the data contains something like `
`, this will ruin the display in the web browser.

When writing scriptlets, it is commonplace to interweave code with HTML tags, for writing convenience. For instance, you can modify the while statement block of the above code to the following:

```
while (rs.next()) {  
%>  
<TR>  
<TD><% out.print(rs.getString(1)); %></TD>  
<TD><% out.print(rs.getString(2)); %></TD>  
<TD><% out.print(rs.getString(3)); %></TD>  
<TD><% out.print(rs.getString(4)); %></TD>  
</TR>  
<%  
}  
This is perfectly legal syntax.
```

Note: Switching from scriptlets to HTML tags and vice versa does not incur any performance penalty. The code in the previous example can therefore be written into as below.

JSP Page that Interweaves HTML Tags with Scriptlets

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    System.out.println("JDBC driver loaded");
}
catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}
%>
<HTML>
<HEAD>
<TITLE>Display All Users</TITLE>
</HEAD>
<BODY>
<CENTER>
<BR><H2>Displaying All Users</H2>
<BR>
<BR>
<TABLE>
<TR>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>User Name</TH>
<TH>Password</TH>
</TR>
<%
String sql = "SELECT FirstName, LastName, UserName, Password" + " FROM Users";
try {
    Connection con = DriverManager.getConnection("jdbc:odbc:JavaWeb");
    Statement s = con.createStatement();
    ResultSet rs = s.executeQuery(sql);
    while (rs.next()) {
%
<TR>
    <TD><% out.print(rs.getString(1)); %></TD>
    <TD><% out.print(rs.getString(2)); %></TD>
    <TD><% out.print(rs.getString(3)); %></TD>
```

```

<TD><% out.print(rs.getString(4)); %></TD>
</TR>
<%
}
rs.close();
s.close();
con.close();
}
catch (SQLException e) {
}
catch (Exception e) {
}
%>
</TABLE>
</CENTER>
</BODY>
</HTML>

```

Scriptlets alone are not sufficient to write efficient and effective JSP pages. As mentioned earlier, you can't declare a method with scriptlets. Additionally, in the above code, the JSP page will try to load the JDBC driver every time the page is requested. This is unnecessary because the page needs to load it only once. How do you run an initialization code as you do in servlets? The answer is explained in the following section: declarations.

Declarations

Declarations allow you to declare methods and variables that can be used from any point in the JSP page. Declarations also provide a way to create initialization and clean-up code by utilizing the `jspInit` and `jspDestroy` methods.

A declaration starts with a `<%!` and ends with a `%>` and can appear anywhere throughout the page. For example, a method declaration can appear above a page directive that imports a class, even though the class is used in the method.

The code given below shows the use of declarations to declare a method called `getSystemTime` and an integer `i`.

Using Declarations to Declare a Method and a Variable

```

<%!
String getSystemTime() {
    return Calendar.getInstance().getTime().toString();
}
%>

```

```

<%@ page import="java.util.Calendar" %>
<%@ page session="false" %>
<%
    out.println("Current Time: " + getSystemTime());
%>
<%! int i; %>

```

The resulting servlet follows. For clarity, some comments have been added to the code.

```
package org.apache.jsp;
```

```

import java.util.Calendar;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.jasper.runtime.*;

public class SimplePage_jsp extends HttpJspBase {
    // method declaration
    String getSystemTime() {
        return Calendar.getInstance().getTime().toString();
    }
    // variable declaration
    int i;
    static {
    }
    public SimplePage_jsp( ) {
    }
    private static boolean _jspx_init = false;
    public final void _jspx_init() throws org.apache.jasper.JasperException { }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException { // the content has been removed
    .
    .
    .
    }
}

```

See how the variable and method are added as a class-level variable and method in the servlet class?

Having the capability to declare a method, The examples given above have been modified to also apply HTML encoding to the data from the database. The modified code is presented below.

JSP Page with a Method Declaration

```
<%!
String encodeHtmlTag(String tag) {
    if (tag==null)
        return null;
    int length = tag.length();
    StringBuffer encodedTag = new StringBuffer(2 * length);
    for (int i=0; i<length; i++) {
        char c = tag.charAt(i);
        if(c=='<')
            encodedTag.append("<");
        else if(c=='>')
            encodedTag.append(">");
        else if(c=='&')
            encodedTag.append("&amp;");
        else if(c=='"')
            encodedTag.append(""");
        //when trying to output text as tag's value as in      // values="???".
        else if(c==' ')
            encodedTag.append("nbsp;");
        else
            encodedTag.append(c);

    }
    return encodedTag.toString();
}
%>
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    System.out.println("JDBC driver loaded");
}
catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}
%>
<HTML>
<HEAD>
<TITLE>Display All Users</TITLE>
</HEAD>
<BODY>
<CENTER>
<BR><H2>Displaying All Users</H2>
```

```

<BR>
<BR>
<TABLE>
<TR>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>User Name</TH>
<TH>Password</TH>
</TR>
<%
String sql = "SELECT FirstName, LastName, UserName, Password" + " FROM Users";
try {
    Connection con = DriverManager.getConnection("jdbc:odbc:JavaWeb");
    Statement s = con.createStatement();
    ResultSet rs = s.executeQuery(sql);

    while (rs.next()) {
%>
<TR>
    <TD><% out.print(encodeHtmlTag(rs.getString(1))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(2))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(3))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(4))); %></TD>
</TR>
<%
}
rs.close();
s.close();
con.close();
}
catch (SQLException e) {
}
catch (Exception e) {
}
%>
</TABLE>
</CENTER>
</BODY>
</HTML>

```

Writing Initialization and Clean-up Code

The code given above suffer from unnecessary repetition: The JSP page tries to load the JDBC driver every time the page is called. This is just a small example. In real life, you may have similar cases where you need to do initialization and clean-up; that is, you want a piece of code to be run only when the JSP servlet is first initialized or when it is destroyed.

You know that every servlet generated from a JSP page must directly or indirectly implement the javax.servlet.jsp.JspPage interface. This interface has two methods: `jspInit` and `jspDestroy`. The JSP container calls the `jspInit` the first time the JSP servlet is initialized and calls the `jspDestroy` when the JSP servlet is about to be removed. These two methods provide a way for initialization and clean-up code. With declarations, you can override these two methods.

You can now modify the previous code by moving the part that loads the JDBC driver to the `jspInit` method. The code is given below.

Utilizing the `jspInit` Method

```
<%!
public void jspInit() {
    try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("JDBC driver loaded");
    } catch (ClassNotFoundException e) {
        System.out.println(e.toString());
    }
}

String encodeHtmlTag(String tag) {
    if (tag==null)
        return null;
    int length = tag.length();
    StringBuffer encodedTag = new StringBuffer(2 * length);
    for (int i=0; i<length; i++) {
        char c = tag.charAt(i);
        if (c=='<')
            encodedTag.append("<");
        else if (c=='>')
            encodedTag.append(">");
        else if (c=='&')
            encodedTag.append("&amp;");
        else if (c=='"')
            encodedTag.append("&quot;");
        //when trying to output text as tag's value as in      // values="???".
        else if (c==' ')
            encodedTag.append("&nbsp;");
        else
            encodedTag.append(c);
    }
    return encodedTag.toString();
} %>
```

```

<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<HTML>
<HEAD>
<TITLE>Display All Users</TITLE>
</HEAD>
<BODY>
<CENTER>
<BR><H2>Displaying All Users</H2>
<BR>
<BR>
<TABLE>
<TR>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>User Name</TH>
<TH>Password</TH>
</TR>
<%
String sql = "SELECT FirstName, LastName, UserName, Password" + " FROM Users";
try {
    Connection con = DriverManager.getConnection("jdbc:odbc:JavaWeb");
    Statement s = con.createStatement();
    ResultSet rs = s.executeQuery(sql);
    while (rs.next()) {
%>
<TR>
    <TD><% out.print(encodeHtmlTag(rs.getString(1))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(2))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(3))); %></TD>
    <TD><% out.print(encodeHtmlTag(rs.getString(4))); %></TD>
</TR>
<%
}
rs.close();
s.close();
con.close();
}
catch (SQLException e) {
}
catch (Exception e) {
} %>

```

```
</TABLE>
</CENTER>
</BODY>
</HTML>
```

The generated servlet is presented here. Note that some parts of the servlet code have been removed for brevity.

```
public class SimplePage_jsp extends HttpJspBase {
    public void jspInit() {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("JDBC driver loaded");
        } catch (ClassNotFoundException e) {
            System.out.println(e.toString());
        }
    }

    String encodeHtmlTag(String tag) {
        .
        .
        .
    }

    static {
    }

    public SimplePage_jsp() {
    }

    private static boolean _jspx_initited = false;
    public final void _jspx_init() throws org.apache.jasper.JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
        .
        .
        .
    }
}
```

Expressions

Expressions are the last type of JSP scripting elements. Expressions get evaluated when the JSP page is requested and their results are converted into a String and fed to the print method of the out implicit object. If the result cannot be converted into a String, an error will be raised at translation time. If this is not detected at translation time, at request-processing time, a ClassCastException will be raised.

An expression starts with a `<%=` and ends with a `%>`. You don't add a semicolon at the end of an expression.

An example of an expression is given below.

Using an Expression

Current Time: `<%= java.util.Calendar.getInstance().getTime() %>`

This expression will be translated into the following statement in the `_jspService` method of the generated servlet:

```
out.print( java.util.Calendar.getInstance().getTime() );
```

The expression given above is equivalent to the following scriptlet:

```
Current Time: <% out.print(java.util.Calendar.getInstance().getTime()); %>
```

As you can see, an expression is shorter because the return value is automatically fed into the `out.print`.

As another example, the while block that you used in previous examples can be re-written as follows using expressions:

```
while (rs.next()) {  
    %>  
    <TR>  
        <TD><%= encodeHtmlTag(rs.getString(1)) %></TD>  
        <TD><%= encodeHtmlTag(rs.getString(2)) %></TD>  
        <TD><%= encodeHtmlTag(rs.getString(3)) %></TD>  
        <TD><%= encodeHtmlTag(rs.getString(4)) %></TD>  
    </TR>  
    <%  
}
```

Standard Action Elements

Standard action elements basically are tags that can be embedded into a JSP page. At compile time, they also are replaced by Java code that corresponds to the predefined task. The following is the list of JSP standard action elements:

- jsp:useBean
- jsp:setProperty
- jsp:getProperty
- jsp:param
- jsp:include
- jsp:forward
- jsp:plugin
- jsp:params
- jsp:fallback

The jsp:useBean, jsp:setProperty, and jsp:getProperty elements are related to Bean and are discussed in next chapter, "Developing JSP Beans." The jsp:param element is used in the jsp:include, jsp:forward, and jsp:plugin elements to provide information in the name/value format, and therefore will be discussed with the three other elements.

jsp:include

The jsp:include action element is used to incorporate static or dynamic resources into the current page. This action element is similar to the include directive, but jsp:include provides greater flexibility because you can pass information to the included resource. The syntax for the jsp:include action element has two forms. For the jsp:include element that does not have a parameter name/value pair, the syntax is as follows:

```
<jsp:include page="relativeURL" flush="true"/>
```

If you want to pass information to the included resource, use the second syntax:

```
<jsp:include page="relativeURL" flush="true"> ( <jsp:param . . . /> )* </jsp:include>
```

The page attribute represents the URL of the included resource in the local server. The flush attribute indicates whether the buffer is flushed. In JSP 1.2, the value of the flush attribute must be true.

In the second form, the * indicates that there can be zero or more elements in the brackets. This means that you also can use this form even though you are not passing any information to the included resource.

jsp:forward

The jsp:forward action element is used to terminate the execution of the current JSP page and switch control to another resource. You can forward control either to a static resource or a dynamic resource.

The syntax for the `jsp:forward` action element has two forms. For the `jsp:forward` element that does not have a parameter name/value pair, the syntax is as follows:

```
<jsp:forward page="relativeURL"/>
```

If you want to pass information to the included resource, use the second syntax:

```
<jsp:forward page="relativeURL"> ( <jsp:param . . . /> )* </jsp:include>
```

The `page` attribute represents the URL of the included resource in the local server.

jsp:plugin

The `jsp:plugin` action element is used to generate HTML `<OBJECT>` or `<EMBED>` tags containing appropriate construct to instruct the browser to download the Java Plugin software, if required, and initiates the execution of a Java applet or a JavaBeans component specified. This is beyond the scope of this book and won't be discussed further.

jsp:params

The `jsp:params` action element can occur only as part of the `<jsp:plugin>` action and will not be discussed in this book.

jsp:fallback

The `jsp:fallback` action element can occur only as part of the `<jsp:plugin>` action and will not be discussed in this book.

Comments

Commenting is part of good programming practice. You can write two types of comments inside a JSP page:

- Comments that are to be displayed in the resulting HTML page at the client browser
- Comments used in the JSP page itself

For comments that are meant to be displayed in the HTML page, you use the comments tags in HTML. This kind of comment must be sent as normal text in a JSP page. For example, the following code sends an HTML comment to the browser:

```
<%
  out.println("<!— Here is a comment —>");
%
>
```

This is equivalent to the following:

```
<!— Here is a comment —>
```

For comments in the JSP page itself, you use the `<%— ... —%>` tag pair.

For example, here is a JSP comment:

```
<%—  
Here is a comment  
—%>
```

A JSP comment can contain anything except the closing tag. For example, this is an illegal comment:

```
<%—  
Here is a comment —%>  
—%>
```

Converting into XML Syntax

XML is getting more and more important in the computing world. JSP pages can be represented using XML, and representing a JSP page as an XML document presents the following benefits:

- The content of the JSP page can be validated against a set of descriptions.
- The JSP page can be manipulated using an XML tool.
- The JSP page can be generated from a textual representation by applying an XML transformation.

Except for the standard action elements, the other JSP programming elements have been presented using the alternative syntax throughout this chapter. This section will show you how non-XML syntax can be converted into XML syntax to share the benefits presented earlier.

Directives

The non-XML syntax for a JSP directive takes the following form:

```
<%@ directive (attribute="value")* %>
```

The XML syntax for a directive is as follows:

```
<jsp:directive:directiveName attribute_list />
```

Scripting Elements

A scripting element can be one of the following: declaration, scriptlet, or expression. The XML syntax for each of the three is given in the following sections.

Declarations

The alternative syntax for a declaration is as follows:

```
<%! declaration code %>
```

This is equivalent to the following XML syntax:

```
<jsp:declaration> declaration code </jsp:declaration>
```

Scriptlets

The alternative syntax for a scriptlet is as follows:

```
<% scriptlet code %>
```

This is equivalent to the following XML syntax:

```
<jsp:scriptlet> scriptlet code </jsp:scriptlet>
```

Expressions

The alternative syntax for an expression is as follows:

```
<%= expression %>
```

This is equivalent to the following XML syntax:

```
<jsp:expression> expression </jsp:expression>
```

Template Data

The `<jsp:text>` XML tag is used to enclose template data in JSP. The syntax is as follows:

```
<jsp:text> text </jsp:text>
```

Brief introduction to JSP Beans

This section presents JavaBeans briefly. You'll find a concise discussion here—it's not meant to be a comprehensive article on the topic.

Really, a bean is just a Java class. You don't need to extend any base class or implement any interface. To be a bean, however, a Java class must follow certain rules specified by the JavaBeans specification. In relation to JavaBeans that can be used from a JSP page, these rules are as follows:

- The bean class must have a no-argument constructor.
- Optionally, a bean can have a public method that can be used to set the value of a property. This method is called a setter method. The method does not return any value, and its name starts with "set" followed by the name of the property. A setter method has the following signature:

```
public void setPropertyName (.PropertyType value);
```

For example, the setter method for the property `operand` must be named `setOperand`. Note that the spelling for the property name is not exactly the same. A property name starts with a lowercase letter, but the name of the setter uses an uppercase letter after "set"; hence `setOperand`.

- Optionally, a bean can have a public method that can be called to obtain the value of a property. This method is called a getter method, and its return type is the same as the property type. Its name

must begin with "get"; followed by the name of the property. A getter method has the following signature:

```
public PropertyType getPropertyname();
```

As an example, the getter method for the property named operand is getOperand. A property name starts with a lowercase letter, but the name of the getter uses an uppercase letter after "get;" hence getOperand. Both setter and getter methods are known as access methods. In JSP, the `jsp:getProperty` and `jsp:setProperty` action elements are used to invoke a getter and a setter method, respectively. You can call these methods the same way you call an ordinary method, however.

Making a Bean Available

Before you can use a bean in your JSP page, you must make the bean available, using the `jsp:useBean` action element. This element has attributes that you can use to control the bean's behavior. The syntax for the `jsp:useBean` element has two forms. The first form is used when you don't need to write any initialization code, and the second form is used if you have Java code that needs to be run when the bean initializes. Code initialization is discussed in the next section.

The two forms of the `jsp:useBean` action element are as follows:

```
<jsp:useBean (attribute="value")/>
```

and

```
<jsp:useBean (attribute="value")> initialization code </jsp:useBean>
```

The `(attribute="value")+` part of the code means that one or more attributes must be present. The five attributes that can be used in a `jsp:useBean` action element are as follows:

- id
- class
- type
- scope
- beanName

Either the class attribute or the type attribute must be present.

The five attributes are explained in the following sections.

id

The id attribute defines a unique identifier for the bean. This identifier can be used throughout the page and can be thought of as the object reference for the bean. The value for the id attribute has the same requirements as a valid variable name in the current scripting language.

class

The class attribute specifies the fully qualified name for the JavaBean class. A fully qualified name is not required if the bean's package is imported using the page directive, however.

type

If the type attribute is present in a jsp:useBean element, it specifies the type of the JavaBean class. The type of the bean could be the type of the class itself, the type of its superclass, or an interface the bean class implements. Normally, this attribute isn't often used.

scope

The scope attribute defines the accessibility and the life span of the bean. This attribute can take one of the following values:

- page
- request
- session
- application

The default value of the scope attribute is page. The scope is a powerful feature that lets you control how long the bean will continue to exist. Each of the attribute values is discussed in the following sections.

page

Using the page attribute value, the bean is available only in the current page after the point where the jsp:useBean action element is used. A new instance of the bean will be instantiated every time the page is requested. The bean will be automatically destroyed after the JSP page loses its scope; namely, when the control moves to another page. If you use the jsp:include or jsp:forward tags, the bean will not be accessible from the included or forwarded page.

request

With the request attribute value, the accessibility of the bean is extended to the forwarded or included page referenced by a jsp:forward or jsp:include action element. The forwarded or included page can use the bean without having a jsp:useBean action element. For example, from inside a forwarded or included page, you can use the jsp:getProperty and jsp:setProperty action elements that reference to the bean instantiated in the original page.

session

A bean with a session scope is placed in the user's session object. The instance of the bean will continue to exist as long as the user's session object exists. In other words, the bean's accessibility extends to other pages.

Because the bean's instance is put in the session object, you cannot use this scope if the page on which the bean is instantiated does not participate in the JSP container's session management. For example, the following code will generate an error:

```
<%@ page session="false" %>
<jsp:useBean id="theBean" scope="session"
    class="com.brainyssoftware.CalculatorBean"/>
```

application

A bean with an application scope lives throughout the life of the JSP container itself. It is available from any page in the application.

beanName

The beanName attribute represents a name for the bean that the instantiate method of the java.beans.Beans class expects.

As an example, the following is a jsp:useBean that instantiates the bean called com.newriders.HomeLoanBean:

```
<jsp:useBean id="theBean" class="com.newriders.HomeLoanBean"/>
```

Alternatively, you can import the package com.newriders using a page directive and refer to the class using its name, as follows:

```
<%@ page import="com.newriders" %>
<jsp:useBean id="theBean" class="HomeLoanBean"/>
```

Note: The bean is available in the page after the jsp:useBean action element. It is not available before that point.

Using JSP Custom Tags

Using JavaBeans, you can separate the presentation part of a JSP page from the business rule implementation (Java code). However, only three action elements—`jsp:useBean`, `jsp:getProperty`, and `jsp:setProperty`—are available for accessing a bean. As such, in some situations, we have to resort to using code in a JSP page. In other words, oftentimes JavaBeans don't offer complete separation of presentation and business rule implementation.

Also, JavaBeans are designed with reusability in mind, meaning that using a bean to output HTML tags directly is not recommended. Outputting HTML tags from a bean makes the Bean useable only from a certain page, even though there are always exceptions to this rule.

In recognition of the imperfection of JavaBeans as a solution to separation of presentation and business rule implementation, JSP 1.1 defined a new feature: custom tags that can be used to perform custom actions.

Custom tags offer some benefits that are not present in JavaBeans. Among others, custom tags have access to all the objects available to JSP pages, and custom tags can be customized using attributes. However, custom tags are not meant to replace JavaBeans completely or make the use of JavaBeans in JSP pages obsolete. JavaBeans have their own fans, too. As you can see, sometimes it is more appropriate to use beans for their reusability, and sometimes it is better to use custom tags. In some circumstances, the choice whether to use JavaBeans or custom tags is not really clear, leaving you with a decision to make based on your experience.

In this chapter, we explore another great feature of JSP: custom tags. It will begin with writing a JSP page that uses custom tags, deploy the small application with Tomcat, and run it in your web browser. After this experience, we start diving deep into the details of the underlying API—the classes and interfaces in the `javax.servlet.jsp.tagext` package. We then create more examples based on those classes and interfaces.

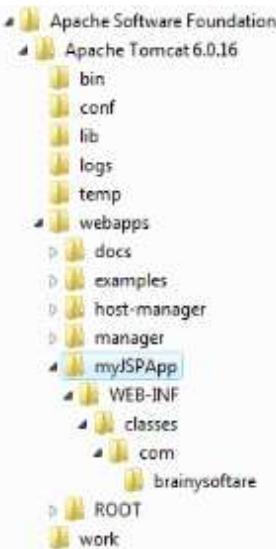
Writing Your First Custom Tag

This section demonstrates the use of custom tags in a JSP page. As usual, you are encouraged to quickly build a small application, deploy it in Tomcat, and call the page from a web browser. For this example, we build a Java component that sends the following String to the browser: "Hello from the custom tag."

This section also presents the step-by-step approach to building a JSP application that uses custom tags.

Before we start developing the application, Following figure shows the directory structure of our JSP application called myJSPApp. You need to complete the directory structure if you have not done so in the previous chapters.

The directory structure of the application.



Now, follow the following five easy steps.

1. Create a TLD file named taglib.tld, as shown below, and save it in the WEB-INF directory.

The TLD File

```
<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd"> <taglib>
<tlibversion>1.0</tlibversion>
<shortname></shortname>
<tag>
<name>myTag</name>
<tagclass>com.brainysoftware.MyCustomTag</tagclass>
</tag>
</taglib>
```

2. Write, compile, and deploy the Java class called MyCustomTag.java given below. Make sure that the .class file is located in the brainysoftware directory, under WEB-INF/classes/com/ directory.

The MyCustomTag.java

```
package com.brainysoftware;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class MyCustomTag extends TagSupport {
    public int doEndTag() throws JspException {
        JspWriter out = pageContext.getOut();
        try {
            out.println("Hello from the custom tag.");
        }
        catch (Exception e) {
        }
        return super.doEndTag();
    }
}
```

3. Create a JSP file from the code given below. Call it SimplePage.jsp and save it under the myJSPApp directory.

The SimplePage.jsp Page

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag/>
```

4. Edit your deployment descriptor (web.xml) file. To use custom tags, you must specify a <taglib> element in your web.xml file. The <taglib> element must appear after the <servlet> and <servlet-mapping> if any. An example of the deployment descriptor is given below.

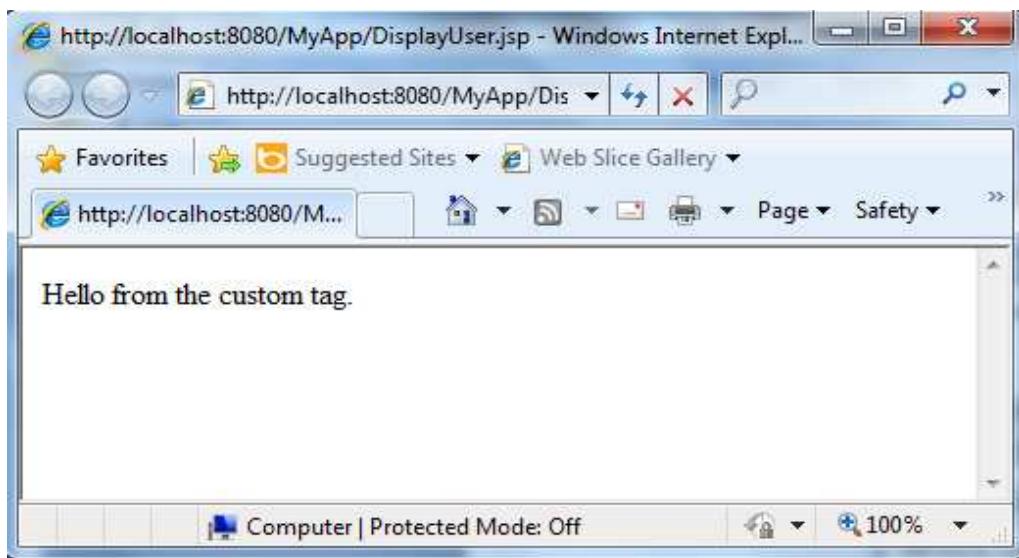
The Deployment Descriptor for This Application

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>template</display-name>
    <taglib>
        <taglib-uri>/myTLD</taglib-uri>
        <taglib-location>/WEB-INF/taglib.tld</taglib-location>
    </taglib>
</web-app>
```

5. Restart Tomcat.

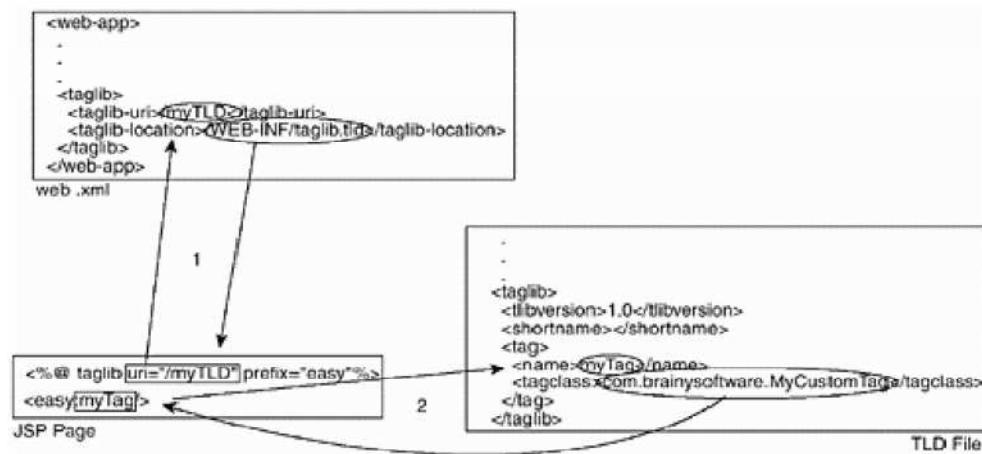
Open your web browser and type <http://localhost:8080/myJSPApp/SimplePage.jsp> in the Address box. You should see something similar to following.

A simple JSP page.



Following figure shows how all the parts work.

The relationship between the deployment descriptor, the JSP page, and the TLD file.



When the user requests the JSP page, the JSP container sees the `<taglib>` tag. Because of this, the JSP container knows that it is seeing a custom tag. It then:

- Consults the deployment descriptor (web.xml) to find the location of the taglib where the URI is `/myTLD`. The deployment descriptor returns the path to the TLD file. The JSP container will remember this path.
- Continues processing the next line and encounters the custom tag `myTag`, prefixed by "easy". Having found out the name and location of the TLD file, the JSP container reads the TLD file and obtains the fully qualified name of the Java class for the tag `myTag`. It reads: `com.brainysoftware.MyCustomTag`.

The JSP container can then load the class for the tag and start processing it.

The Role of the Deployment Descriptor

The preceding example illustrates how the JSP container can find the name and location of the TLD file by looking up the deployment descriptor. Even though this is a common practice, you can eliminate the role of the deployment descriptor by writing the name and location of the TLD file directly on the JSP page. Therefore, the code given below is also valid.

An Alternative JSP Page for the Application

```
<%@ taglib uri="/WEB-INF/taglib.tld" prefix="easy"%>
<easy:myTag/>
```

However, directly referring to the TLD file lacks the flexibility to move and change the name of the TLD file without having to edit all the JSP pages that utilize it.

Therefore, indirect reference of the TLD file involving the deployment descriptor is preferable.

Note that in all examples in this chapter, the taglib tag's URI attribute is assigned to /myTLD. It is assumed you have edited the web.xml file accordingly.

The Tag Library Descriptor

A tag library descriptor (TLD) file is an XML document that defines a tag library and its tags. The first lines of a TLD file are the standard XML header information, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

A TLD file contains the <taglib> root element. This element can have the following subelements:

- tlibversion
- jspversion
- shortname
- info
- uri
- tag

Of these six elements, the tag element is the only one that can have attributes. However, the tlibversion, shortname, and tag are required in the TLD file.

The tlibversion element specifies the version number of the tag library in the following format:

$([0-9])^* ("." [0-9])? ("." [0-9])? ("." [0-9])?$

The asterisk means that the values in the bracket can be repeated zero or more times, and the question mark means that the value in the bracket is optional. The following are some valid values for a tlibversion element: 1.1.1.1, 2, 2.3, 3.3.6, and so on.

The `jspversion` element specifies the JSP version. The format is the same as the `tlibversion` element.

`([0-9])*("." [0-9])?("." [0-9])?("." [0-9])?`

The `shortname` element specifies a short name for the tag library. The value for this element must begin with a letter and must not contain blank space.

The `info` element contains the information used for documentation purposes. Its default value is an empty string.

The `uri` element specifies the link to an additional source of documentation for the tag library.

The `tag` element is the most important element in a tag library. You can specify more than one `tag` element in the same TLD file. Therefore, you only have one TLD file for each JSP application.

The `tag` element is explained in more detail in the following section.

The following is an example of a TLD file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"> <taglib>
<tlibversion>1.0.123.123.234</tlibversion> <jspversion>1.2</jspversion>
<shortname></shortname>
<info>Example tags</info>
<uri></uri>
<tag>
<name>myTag</name>
<tagclass>com.brainysoftware.MyCustomTag</tagclass>
<attribute>
<name>number</name>
<required>true</required>
</attribute>
<attribute>
<name>power</name>
<required>true</required>
</attribute>
</tag>
</taglib>
```

The tag Element

The tag element specifies a custom tag in the library. This element can have six subelements:

- name
- tagclass
- teiclass
- bodycontent
- info
- attribute

Of these six tag subelements, name and tagclass are mandatory.

The name element specifies an identifier for the tag.

The tagclass element specifies the fully qualified name of the Java class that handles the processing of this tag.

The teiclass element specifies the helper class for this tag, if there is one. tei is short for TagExtraInfo, the name of a class in the javax.servlet.jsp.tagext package. TagExtraInfo is explained in more detail later in this chapter when we discuss the API.

The bodycontent element specifies the type of the body content of the tag, if any. A body content is what appears between the opening and closing tags. This element can have one of the following values: empty, JSP, or tagdepen-dent. The empty value means that there is no body content supported by the tag. Assigning "JSP" as the value of the bodycontent element indicates that the body content values are one or more JSP elements. The last value, tagdepen-dent, indicates that the tag's body is to be interpreted by the tag.

The info element of the tag element contains an informational description.

The attribute element specifies zero or more attributes. The attribute element can have three subelements: name, required, and rtextprvalue. Only name is a required subelement of the attribute.

The name subelement is the identification for the attribute. The required subelement indicates whether the attribute is mandatory and can take a value of true or false, with false as the default value.

The rtextprvalue subelement indicates whether the value for this attribute can be determined at request time. If the value of this subelement is true, you can assign a request time value such as a JSP expression.

The Custom Tag Syntax

To use a custom tag in a JSP page, you first need to use a taglib directive in your page. The taglib directive has the following syntax:

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

The uri attribute specifies an absolute or relative URI that uniquely identifies the tag library descriptor associated with this prefix.

The prefix attribute defines a string that will become the prefix to distinguish a custom action. With a taglib directive, you then can use a custom tag of the following format for a custom tag that does not have a content body:

```
<prefix:tagName/>
```

Or, you can use the following format for a custom tag that has a content body:

```
<prefix:tagName>body</prefix:tagName>
```

You can pass attributes to the tag handler by specifying the attribute(s) in the custom tag, each with the following format:

```
attributeName="attributeValue"
```

The following example is a custom tag whose prefix is m and whose name is myTag. The tag has two attributes: number with a value of 12, and power with a value of 13.

```
<m:myTag number="12" power="13"/>
```

Note: An attribute value must be quoted.

The JSP Custom Tag API

A tag handler is the Java class that is linked to a custom tag and gets invoked every time the JSP container encounters the custom tag. The tag handler is so named because it handles the processing of the tag. To be functional, a tag handler must implement an interface in the javax.servlet.jsp.tagext package or extend one of the classes in the same package.

In JSP 1.2, the javax.servlet.jsp.tagext has four interfaces and twelve classes, four more members than the same package in the JSP 1.1 specification. Some of the members of the javax.servlet.jsp.tagext package are explained here.

The two most important interfaces are Tag and BodyTag. These two interfaces also set the life cycle of a tag handler. Either Tag or BodyTag must be implemented by a tag handler, either directly or indirectly.

The Tag Interface

Tag handlers that implement the Tag interface must provide implementations for all the interface's methods. These methods are as follows:

- doStartTag
- doEndTag
- getParent

- setParent
- setPageContext
- release

You can write a tag handler by simply implementing the Tag interface and providing blank implementations of all its six methods. For example, the code given below is a tag handler called BasicTagHandler that does not do anything.

A Simple Tag Handler that Does Not Do Anything

```
package com.brainysoftware;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class BasicTagHandler implements Tag {
    public void setParent(Tag t) {
    }
    public void setPageContext(PageContext p) {
    }
    public void release() {
    }
    public Tag getParent() {
        return null;
    }
    public int doStartTag() {
        return EVAL_BODY_INCLUDE;
    }
    public int doEndTag() {
        return EVAL_PAGE;
    }
}
```

Note that both doStartTag and doEndTag return an integer. Their return values are one of the four static final integers defined in the Tag interface: SKIP_BODY, EVAL_BODY_INCLUDE, SKIP_PAGE, and EVAL_PAGE. Tag.SKIP_BODY and Tag.EVAL_BODY_INCLUDE are valid return values of the doStartTag method, whereas Tag.SKIP_PAGE and Tag.EVAL_PAGE are valid return values of the doEndTag method.

The Life Cycle of a Tag Handler

The life cycle of a tag handler that implements the Tag interface is controlled by the JSP container by calling the methods in the following order:

1. The JSP container obtains an instance of the tag handler from the pool or creates a new one. It then calls the setPageContext, passing a PageContext object representing the JSP page where the

custom tag is found. If you want to access the JSP page, you need to assign this PageContext object to an object reference that has class scope. The setPageContext method has the following signature:

public void setPageContext(PageContext pageContext)

2. The JSP container then calls the setParent method. This method passes a tag object, which represents the closest tag enclosing the current tag handler. If there is no enclosing tag, a null object reference is passed. The signature of the setParent method is as follows:

public void setParent(Tag parent)

3. The JSP container then sets all the attributes in the custom tag, if any. Attributes are handled like properties in a JavaBean, namely by using the getter and setter methods. For example, if the custom tag has an attribute named temperature, the getter is called getTemperature and the setter is called setTemperature. The JSP container calls all the available setter methods to set attribute values.

4. Next, the JSP container calls the doStartTag, whose signature is as follows: `public int doStartTag() throws javax.servlet.jsp.JspException` The doStartTag method can return either Tag.SKIP_BODY and Tag.EVAL_BODY_INCLUDE. If Tag.SKIP_BODY is returned, the JSP container will not process the tag's body contents, if any. If the doStartTag method returns Tag.EVAL_BODY_INCLUDE, the body contents, if any, will be processed normally.

5. Regardless of the return value of the doStartTag method, the JSP container next calls the doEndTag method. This method has the following signature: `public int doEndTag() throws javax.servlet.jsp.JspException` The doEndTag method returns either Tag.SKIP_PAGE or Tag.EVAL_PAGE. If Tag.SKIP_PAGE is returned, the JSP container will not process the remaining of the JSP page. If Tag.EVAL_PAGE is returned, the JSP container processes the rest of the JSP page as normal.

6. The release method is the last method that the JSP container calls. This method has the following signature:

public void release()

You should write any clean-up code in this method implementation. For example, you might want to close a file that was opened in the other method or destroy a database connection.

7. The JSP container returns the instance of the tag handler to a pool for future use.

A Content Substitution Tag Handler

The following example illustrates a tag handler that provides implementations for the setPageContext and doStartTag methods. The setPageContext assigns the PageContext object passed in by the JSP container to an object reference.

The doStartTag then obtains the JspWriter object through the getOut method of the PageContext and writes something to it. Because the tag handler writes directly to the JspWriter object of the

current JSP page, whatever it writes is sent to the web browser. The tag handler is called MyCustomTag and is shown below.

The Content Substitution Tag Handler

```
package com.brainysoftware;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class MyCustomTag implements Tag {
    PageContext pageContext;

    public void setParent(Tag t) {
    }

    public void setPageContext(PageContext p) {
        pageContext = p;
    }

    public void release() {
    }

    public Tag getParent() {
        return null;
    }

    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.println("Hello from the tag handler.");
        } catch(Exception e) {
        }
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }
}
```

To test the tag handler, you need a JSP page and a TLD file as given below.

The JSP Page that Contains a Custom Tag that Uses the Content Substitution Tag Handler

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag/>
```

The TLD File for This Example

```
<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"> <taglib>
<tlibversion>1.0</tlibversion>
<shortname></shortname>
<tag>
<name>myTag</name>
<tagclass>com.brainysoftware.MyCustomTag</tagclass>
</tag>
</taglib>
```

An Example: Using Attributes in a custom Tag

The following example presents a tag handler named DoublerTag that doubles an integer and outputs the result to the web browser. It demonstrates the use of an attribute called number. The tag handler has a setter called setNumber that is called by the JSP container to set the attribute value. The code for the tag handler is given below.

The DoublerTag

```
package com.brainysoftware;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DoublerTag implements Tag {
    private int number;
    public void setNumber(int number) {
        this.number = number;
    }
    PageContext pageContext;

    public void setParent(Tag t) {
    }

    public void setPageContext(PageContext p) {    pageContext = p;
    }
}
```

```

public void release() {
}

public Tag getParent() {
    return null;
}
public int doStartTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.println("Double of " + number + " is " + (2 * number));    }
    catch(Exception e) {
    }
    return EVAL_BODY_INCLUDE;
}

public int doEndTag() throws JspException {
    return EVAL_PAGE;
}
}

```

For the code to run, you need a JSP page and a TLD file as given below.

The JSP Page that Calls the DoublerTag

```

<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag number="12"/>

```

The TLD File for the Example

```

<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"> <taglib>
<tlibversion>1.0</tlibversion>
<shortname></shortname>
<tag>
<name>myTag</name>
<tagclass>com.brainysoftware.DoublerTag</tagclass>
<attribute>
<name>number</name>
<required>true</required>
</attribute>
</tag>
</taglib>

```

The IterationTag Interface

The IterationTag interface extends the Tag interface by adding a new method called doAfterBody and a static final integer EVAL_BODY_AGAIN. This method is invoked after the doStartTag method and can return either the Tag.SKIP_BODY or IterationTag.EVAL_BODY_AGAIN. If the latter is returned, the doAfterBody is called again. If the return value is Tag.SKIP_BODY, the body will be skipped and the JSP container will call the doEndTag method.

An Example: A PowerTag

The following example demonstrates a tag handler that implements the IterationTag interface and illustrates the use of the doAfterBody method. The tag handler, called PowerTag, calculates a base number raised to the power of another number. It then writes to the Web browser the following:

number^{power}=result

For instance, if you pass 2 for number and 3 for power, the tag handler returns the following String to the browser:

2³=8

The code of the tag handler is given below.

The PowerTag Tag Handler

```
package com.brainysoftware;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class PowerTag implements IterationTag { PageContext pageContext;

    private int number;
    private int power;
    private int counter;
    private int result = 1;

    // the setter for number
    public void setNumber(int number) {
        this.number = number;
    }

    // the setter for power
    public void setPower(int power) {
        this.power = power;
    }

    public void doStartTag() throws JspException {
        counter = 1;
        result = 1;
    }

    public Object doAfterBody() throws JspException {
        if (counter < power) {
            result *= number;
            counter++;
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }

    public void doEndTag() throws JspException {
        pageContext.getOut().print(result);
    }
}
```

```

public void setParent(Tag t) {
}

public void setPageContext(PageContext p) {
    pageContext = p;
}

public void release() {
}

public Tag getParent() {
    return null;
}

public int doStartTag() {
    return EVAL_BODY_INCLUDE;
}

public int doAfterBody() {
    counter++;
    result *= number;
    if(counter==power)
        return SKIP_BODY;
    else
        return EVAL_BODY_AGAIN;
}

public int doEndTag() throws JspException {
    System.out.println("doEndTag");
    try {
        JspWriter out = pageContext.getOut();
        out.println(number + "^" + power + "=" + result);      }
    catch(Exception e) {
    }
    return EVAL_PAGE;
}
}

```

To test the code, you need a JSP page and a TLD file as given below.

The JSP Page that Calls PowerTag

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag number="2" power="3"/>
```

The TLD File that Is Used in the Example

```
<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"> <taglib>
<tlibversion>1.0</tlibversion>
<shortname></shortname>
<tag>
  <name>myTag</name>
  <tagclass>com.brainysoftware.PowerTag</tagclass>
  <attribute>
    <name>number</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>power</name>
    <required>true</required>
  </attribute>
</tag>
</taglib>
```

Manipulating Body Contents with BodyTag and BodyContent

A JSP custom tag can have a body content, such as the following tag:

```
<%@ taglib uri="/myTLD" prefix="x"%>
<x:theTag>This is the body content</x:theTag>
```

When you use the Tag and IterationTag interfaces, you cannot manipulate the body content. You don't even have access to it.

If you want to manipulate the body content of the custom tag, you have to use the BodyTag interface and the BodyContent class in the javax.servlet.jsp.tagext package. This section introduces you to these two elements and builds an example of a tag handler that manipulates the body content.

The BodyTag Interface

The BodyTag interface extends the IterationTag interface by adding two methods, doInitBody and setBodyContent, and two static final integers, EVAL_BODY_BUFFERED and EVAL_BODY_TAG. As of JSP 1.2, however, EVAL_BODY_TAG is deprecated. A tag handler that implements the BodyTag interface has a life cycle similar to the ones implementing the

IterationTag interface. The difference is that the doStartTag method of the tag handler implementing BodyTag can return SKIP_BODY, EVAL_BODY_INCLUDE, or EVAL_BODY_BUFFERED. If the method returns EVAL_BODY_INCLUDE, the body is evaluated as it is in IterationTag. If the method returns EVAL_BODY_BUFFERED, a BodyContent object is created that represents the custom tag's body content. You learn more about the BodyContent class later in this section.

Two extra methods are called by the JSP container in tag handlers implementing the BodyTag interface: setBodyContent and doInitBody. The two methods have the following signatures:

```
public void setBodyContent(BodyContent bodyContent)  
public void doInitBody() throws javax.servlet.jsp.JspException
```

The setBodyContent method is called after the doStartTag, followed by the doInitBody method. The setBodyContent will not be invoked, however, if one or both of the following is true:

- The custom tag does not have a body content.
- The custom tag has a body content but the doStartTag method returns SKIP_BODY or EVAL_BODY_INCLUDE.

The doInitBody method can be used to prepare for evaluation of the body. Normally, this method is called by the JSP container after the setBody Content method. This method will not be called, however, if one of the following is true:

- The custom tag does not have a body content.
- The custom tag has a body content, but the doStartTag method returns SKIP_BODY or EVAL_BODY_INCLUDE.

The BodyContent Class

The BodyContent class is an abstract class that extends the javax.servlet.jsp.JspWriter class. The BodyContent class represents the body content of the custom tag, if any. You obtain the body content from the setBodyContent method in the BodyTag interface.

An Example: Manipulating the Body Content

The following example shows a tag handler that manipulates the body content of a JSP custom tag. The tag handler does two things: HTML encode the body content and print the encoded version of the body content to the browser. The tag handler is called EncoderTag and incorporates the htmlEncodeTag method from the com.brainysoftware.java.StringUtil class. The tag handler is given in below.

The EncoderTag

```
package com.brainysoftware;  
  
import javax.servlet.jsp.*;  
import javax.servlet.jsp.tagext.*;
```

```

public class EncoderTag implements BodyTag {
    PageContext pageContext;
    BodyContent bodyContent;

    /**
     * Encode an HTML tag so it will be displayed      * as it is on the browser.
     * Particularly, this method searches the
     * passed in String and replace every occurrence      * of the following characters:
     * '<' with "<"
     * '>' with ">"
     * '&' with "&"
     * "" with """
     * ' ' with " "
     */
    private String encodeHtmlTag(String tag) {
        if (tag==null)
            return null;
        int length = tag.length();
        StringBuffer encodedTag = new StringBuffer(2 * length);
        for (int i=0; i<length; i++) {
            char c = tag.charAt(i);
            if(c=='<')
                encodedTag.append("<");
            else if(c=='>')
                encodedTag.append(">");
            else if(c=='&')
                encodedTag.append("&");
            else if(c=='"')
                encodedTag.append(""");
            //when trying to output text as tag's value as in      // values="???".
            else if(c==' ')
                encodedTag.append(" ");
            else
                encodedTag.append(c);
        }
        return encodedTag.toString();
    }

    public void setParent(Tag t) {
    }

    public void setPageContext(PageContext p) {

```

```

    pageContext = p;
}

public void release() {
}

public Tag getParent() {
    return null;
}

public int doStartTag() {
    return EVAL_BODY_BUFFERED;
}

public void setBodyContent(BodyContent bodyContent) {
    this.bodyContent = bodyContent;
}

public void doInitBody() {

}

public int doAfterBody() {

    String content = bodyContent.getString();
    try{
        JspWriter out = bodyContent.getEnclosingWriter();
        out.print(encodeHtmlTag(content));
    }
    catch(Exception e) {}

    return SKIP_BODY;
}

public int doEndTag() throws JspException {
    return EVAL_PAGE;
}
}

```

The important methods are the `setBodyContent` and the `doAfterBody`. The `setBodyContent` method passes the `BodyContent` objects from the JSP container to the `bodyContent` object reference that has class-scope, as shown here:

```
public void setBodyContent(BodyContent bodyContent) {
```

```
this.bodyContent = bodyContent; }
```

You cannot manipulate the body content in the setBodyContent method. You must wait until the doInitBody is called. You write the following manipulation code in the doAfterBody method to take care of this:

```
String content = bodyContent.getString();
try{
    JspWriter out = bodyContent.getEnclosingWriter();
    out.print(encodeHtmlTag(content));
}
catch(Exception e) { }

return SKIP_BODY;
```

First, you obtain the String representation of the body content using the getString method of the BodyContent class. Next, to write to the browser, you first need to use the getEnclosingWriter method to obtain the implicit object out of the current JSP page. Finally, to output a String to the browser, you call the print method of JspWriter. To use the tag handler, you need the JSP page and a TLD file, as shown below.

The JSP Page that Uses the EncoderTag Tag Handler

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag><BR> means change line</easy:myTag>
```

The TLD File for this Example

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <shortname></shortname>
    <tag>
        <name>myTag</name>
        <tagclass>com.brainysoftware.EncoderTag</tagclass>
        <bodycontent>tagdependent</bodycontent>
    </tag>
</taglib>
```

The Support Classes

Although the three interfaces—Tag, IterationTag, and BodyTag—provide a great way to write tag handlers, one apparent drawback is inherent to all interfaces: You must provide implementations for all the methods, including those you don't use. This, of course, makes the code look more complex than necessary, and the class takes longer to write and debug.

To solve this problem, the javax.servlet.jsp.tagext package provides support classes that implement those interfaces. These classes are TagSupport and BodyTagSupport. Now, instead of implementing an interface, you can extend one of these classes, as this section explains.

The TagSupport Class

The TagSupport class implements the IterationTag interface. Its signature is as follows: public class TagSupport implements IterationTag, java.io.Serializable The TagSupport class is intended to be used as the base class for tag handlers.

The BodyTagSupport Class

The BodyTagSupport class implements the BodyTag interface and has the following signature:

```
public class BodyTagSupport extends TagSupport implements BodyTag
```

This class is meant to be subclassed by tag handlers that need to implement the BodyTag interface.

To illustrate the usefulness of these support classes, the following example shows a tag handler called CapitalizerTag that converts a body content to its uppercase version and outputs it to the browser. The code is given below.

The CapitalizerTag

```
package com.brainysoftware;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class CapitalizerTag extends BodyTagSupport {
    public int doAfterBody() {
        String content = bodyContent.getString();
        try{
            JspWriter out = bodyContent.getEnclosingWriter();
            out.print(content.toUpperCase());
        }
        catch(Exception e){}
        return SKIP_BODY;
    }
}
```

See how simple the tag handler has become? You don't need to provide implementations of methods you don't use.

To complete the example, here is the JSP page that uses the CapitalizerTag tag handler and the TLD file.

The JSP Page that Uses the CapitalizerTag

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag>See the big picture?</easy:myTag>
```

The TLD File for this Example

```
<?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <shortname></shortname>
  <tag>
    <name>myTag</name>
    <tagclass>com.brainysoftware.CapitalizerTag</tagclass>
    <bodycontent>tagdependent</bodycontent>
  </tag>
</taglib>
```

Working with JSTL

The JSP Standard Template Library (JSTL) is a very new component released by Sun for JSP programming. JSTL allows you to program your JSP pages using tags, rather than the scriptlet code that most JSP programmers are already accustomed to. JSTL can do nearly everything that regular JSP scriptlet code can do.

JSTL was introduced to allow JSP programmers to program using tags rather than Java code. To show why this is preferable, a quick example is in order. We will examine a very simple JSP page that counts to ten. We will examine this page both as regular scriptlet-based JSP, and then as JSTL. When the count to ten example is programmed using scriptlet based JSP, the JSP page appears as follows.

```
<html>
  <head>
    <title>Count to 10 in JSP scriptlet</title>
  </head>
  <body>
<%
for(int i=1;i<=10;i++)
{
%>
<%=i%><br/>
<%
} %>
</body>
</html>
```

As you can see from the preceding example, using scriptlet code produces page source code that contains a mix of HTML tags and Java statements. There are several reasons why this mixing of programming styles is not optimal.

The primary reason that it is not optimal to mix scriptlet and tag-based code is readability. This readability applies both to humans and computers. JSTL allows the human programmer to look at a program that consists entirely of HTML and HTML-like tags.

The readability of JSP scriptlet code does not just apply to human beings. The mixing of scriptlet and HTML code is also hard for computers to read. This is particularly true of HTML authoring tools such as Someone's Dreamweaver and Microsoft FrontPage. Currently, most HTML authoring tools will segregate JSP scriptlet code as non-editable blocks. The HTML authoring tools usually do not modify the JSP scriptlet code directly.

The following code shows how the count to ten example would be written using JSTL. As you can see, this code listing is much more constant, as only tags are used. HTML and JSTL tags are mixed to produce the example.

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %> <html>
<head>
<title>Count to 10 Example (using JSTL)</title> </head>

<body>
<c:forEach var="i" begin="1" end="10" step="1">
<c:out value="${i}" />
<br />
</c:forEach>
</body>
</html>
```

When you examine the preceding source code, you can see that the JSP page consists entirely of tags. The above code makes use of HTML tags such `<head>` and `
`. The use of tags is not confined just to HTML tags. This code also makes use of JSTL tags such as `<c:forEach>` and `<c:out>`.

The JSTL Tag Libraries

JSTL is often spoken of as a single-tag library. Actually, JSTL consists of four tag libraries. These tag libraries are summarized as follows.

Core Tag Library: Contains tags that are essential to nearly any Web application. Examples of core tag libraries include looping, expression evaluation, and basic input and output.

Formatting/Internationalization Tag Library: Contains tags that are used to format and parse data. Some of these tags will parse data, such as dates, differently based on the current locale.

Database Tag Library: Contains tags that can be used to access SQL databases. These tags are normally used only to create prototype programs. This is because most programs will not handle database access directly from JSP pages. Database access should be embedded in EJBs that are accessed by the JSP pages.

XML Tag Library: Contains tags that can be used to access XML elements. Because XML is used in many Web applications, XML processing is an important feature of JSTL.

Now, it's time to take a very brief look at a few of the core tags. We will examine a simple example that shows how to process data that a user enters into a form. Before we examine this program, we must first see how JSTL handles expressions. Expression handling in JSTL is accomplished by using the EL expression language, just as it is done in JSP 2.0.

The EL Expression Language

One major component of JSP 2.0 is the new expression language named EL. EL is used extensively in JSTL. However, it is important to remember that EL is a feature of JSP and not of JSTL. JSP scriptlet code used with JSP 2.0 can contain EL expressions. The following lines of code demonstrate using EL inside of JSP scriptlet code.

```
<p> Your total, including shipping is ${total+shipping} </p>
```

As you can see from the preceding code, the values "total" and "shipping" are added and displayed as the HTML is generated. These expressions can be used inside of JSTL tags as well. One important requirement of JSTL 1.0 was that JSTL could be used with JSP 1.2. Because JSP 1.2 does not support EL, it is necessary to provide a few additional JSTL tags that facilitate the use of EL. For example, if you wanted to use JSTL to display the above expression, you would use the following code.

```
<p> Your total, including shipping is <c:out var="${total+shipping}" /> </p>
```

One of the requirements of JSTL was that it not require JSP 2.0 to run. By providing a tag that is capable of displaying EL expressions, this requirement is met.

JSTL Example

We will now examine a simple example that uses JSTL. For this example, we will examine a common procedure that is done by many Web applications. We will see how to POST a form and process the results from that POST. A simple program that is capable of doing this is shown below.

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>If with Body</title>
  </head>

  <body>
    <c:if test="${pageContext.request.method=='POST'}">
      <c:if test="${param.guess=='Java'}">You guessed it!
      <br />    <br />    <br />
    </c:if>
```

```

<c:if test="${param.guess!='Java'}">You are wrong
    <br />    <br />    <br />
</c:if>
</c:if>

<form method="post">Guess what computer language I am thinking of?
    <input type="text" name="guess" />
    <input type="submit" value="Try!" />    <br />
</form>
</body>
</html>

```

This simple Web page will display a form and ask the user to guess what computer language the program is thinking of. Of course, the computer is thinking of "Java." This page begins by checking to see if a POST was done. This allows both the form, and the code that handles the form, to be placed on one single page. This is done with the following JSTL if statement.

```
<c:if test="${pageContext.request.method=='POST'}">
```

Here you can see that the `<c:if>` tag uses an EL expression to evaluate whether the request method is POST. If data was posted to the page, the value that the user entered for their guess is stored in a parameter named "guess". This is because "guess" was specified as the name of the form input item. We must now check to see whether this parameter is equal to the word "Java". This is done with the following `<c:if>` tag.

```

<c:if test="${param.guess=='Java'}">
    You guessed it!
</c:if>

```

As you can see, the body of the `<c:if>` tag is executed if the statement evaluates to true. In this article, we began to examine the basics of how JSTL is installed and how it works.

The core tags of JSTL also include tags for looping, iteration, and variable handling. By using these tags, you can iterate through collections, access user session data, and perform other core tasks that all Web applications perform. In addition to the core tag library, the XML, database, and formatting tag libraries are also provided for more advanced uses.

Following is a detailed description of Expression Language:

Expressions (EL)

In Attribute Values

- `<a:tag value="${expr}" />`
- `<a:tag value="me${expr} ${expr}" />`

Bean Property Access

bean.name

bean["name"]

Indexed Property Access

bean.property[index]

Map Property Access

bean.property["key"]

Deactivating Expression Evaluation

Because the pattern that identifies EL expressions--\${ }--was not reserved in the JSP specifications before JSP 2.0, there may be applications where such a pattern is intended to pass through verbatim. To prevent the pattern from being evaluated, you can deactivate EL evaluation.

To deactivate the evaluation of EL expressions, you specify the *isELIgnored* attribute of the *page* directive:

```
<%@ page isELIgnored ="true/false" %>
```

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

The default value varies depending on the version of the web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backward compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. You can also deactivate EL expression evaluation for a group of JSP pages.

Using Expressions

EL expressions can be used:

- In static text
- In any standard or custom tag attribute that can accept an expression

The value of an expression in static text is computed and inserted into the current output. If the static text appears in a tag body, note that an expression will not be evaluated if the body is declared to be tagdependent.

There are three ways to set a tag attribute value:

1. With a single expression construct:

```
<some:tag value="${expr}" />
```

The expression is evaluated and the result is coerced to the attribute's expected type.

2. With one or more expressions separated or surrounded by text:

```
<some:tag value="some${expr}${expr}text${expr}">
```

The expressions are evaluated from left to right. Each expression is coerced to a String and then concatenated with any intervening text. The resulting String is then coerced to the attribute's expected type.

3. With text only:

```
<some:tag value="sometext">
```

In this case, the attribute's String value is coerced to the attribute's expected type.

Expressions used to set attribute values are evaluated in the context of an expected type. If the result of the expression evaluation does not match the expected type exactly, a type conversion will be performed. For example, the expression \${1.2E4} provided as the value of an attribute of type float will result in the following conversion:

```
Float.valueOf("1.2E4").floatValue()
```

Variables

The web container evaluates a variable that appears in an expression by looking up its value according to the behavior of `PageContext.getAttribute(String)`. For example, when evaluating the expression \${product}, the container will look for product in the page, request, session, and application scopes and will return its value. If product is not found, null is returned. A variable that matches one of the implicit objects described in [Implicit Objects](#) will return that implicit object instead of the variable's value.

Properties of variables are accessed using the `.` operator and can be nested arbitrarily.

The JSP expression language unifies the treatment of the `.` and `[]` operators. `expr-a.identifier-b` is equivalent to `expr-a["identifier-b"]`; that is, the expression `expr-b` is used to construct a literal whose value is the identifier, and then the `[]` operator is used with that value.

To evaluate `expr-a[expr-b]`, evaluate `expr-a` into `value-a` and evaluate `expr-b` into `value-b`. If either `value-a` or `value-b` is null, return null.

If `value-a` is a Map, return `value-a.get(value-b)`. If `!value-a.containsKey(value-b)`, then return null.

If `value-a` is a List or array, coerce `value-b` to int and return `value-a.get(value-b)` or `Array.get(value-a, value-b)`, as appropriate. If the coercion couldn't be performed, an error is returned. If the get call returns an `IndexOutOfBoundsException`, null is returned. If the get call returns another exception, an error is returned.

If value-a is a JavaBeans object, coerce value-b to String. If value-b is a readable property of value-a, then return the result of a get call. If the get method throws an exception, an error is returned.

Implicit Objects

The JSP expression language defines a set of implicit objects:

pageContext:	The context for the JSP
page:	Provides access to various objects including:
servletContext:	The context for the JSP page's servlet and any web components contained in the same application.
session:	The session object for the client.
request:	The request triggering the execution of the JSP page.
response:	The response returned by the JSP page.

In addition, several implicit objects are available that allow easy access to the following objects:

param:	Maps a request parameter name to a single value
paramValues:	Maps a request parameter name to an array of values
header:	Maps a request header name to a single value
headerValues:	Maps a request header name to an array of values
cookie:	Maps a cookie name to a single cookie
initParam:	Maps a context initialization parameter name to a single value

Finally, there are objects that allow access to the various scoped variables described in Using Scope Objects.

pageScope:	Maps page-scoped variable names to their values
requestScope:	Maps request-scoped variable names to their values
sessionScope:	Maps session-scoped variable names to their values
applicationScope:	Maps application-scoped variable names to their values

When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example, \${pageContext} returns the PageContext object, even if there is an existing pageContext attribute containing some other value.

Literals

The JSP expression language defines the following literals:

Boolean:	true and false
Integer:	as in Java
Floating point:	as in Java
String:	with single and double quotes; " is escaped as \" , ' is escaped as \' , and \ is escaped as \\.
Null:	null

Operators

In addition to the . and [] operators discussed in Variables, the JSP expression language provides the following operators:

Arithmetic:	+,- (binary), *, / and div, % and mod, - (unary)
Logical:	and, &&, or, , not, !
Relational:	==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values, or against boolean, string, integer, or floating point literals.
Empty:	The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
Conditional:	A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

The precedence of operators highest to lowest, left to right is as follows:

[] .
() - Used to change the precedence of operators.
- (unary) not ! empty
* / div % mod
+ - (binary)
<> <= >= lt gt le ge
== != eq ne
&& and
|| or
?:

Reserved Words

The following words are reserved for the JSP expression language and should not be used as identifiers.

and eq gt true instanceof

or ne le false empty

not lt ge null div mod

Note that many of these words are not in the language now, but they may be in the future, so you should avoid using them.

Examples

EL Expression	Result
<code> \${1 > (4/2)}</code>	false
<code> \${ 4.0 >= 3 }</code>	true
<code> \${ 100.0 == 100 }</code>	true
<code> \${ (10*10) ne 100 }</code>	false
<code> \${ 'a' < 'b' }</code>	true

<code> \${'hip' gt 'hit'}</code>	true
<code> \${4 > 3}</code>	true
<code> \${1.2E4 + 1.4}</code>	12001.4
<code> \${3 div 4}</code>	0.75
<code> \${10 mod 4}</code>	2
<code> \${empty param.Add}</code>	true is the request parameter named Add is null or an empty string
<code> \${requestScope['msg']} or \${requestScope.msg}</code>	The value of the request-scoped attribute named msg

Functions

The JSP expression language allows you to define a function that can be invoked in an expression. Functions are defined using the same mechanisms as custom tags.

Using Functions

Functions can appear in static text and tag attribute values.

To use a function in a JSP page, you use a taglib directive to import the tag library containing the function. Then you preface the function invocation with the prefix declared in the directive.

For example, the date example page index.jsp imports the /functions library and invokes the function equals in an expression:

```
<%@ taglib prefix="f" uri="/functions"%>
...
<c:when test="${f:equals(selectedLocaleString,localeString)}" >
```

Defining Functions

To define a function you program it as a public static method in a public class. The mypkg.MyLocales class in the date example defines a function that tests the equality of two Strings as follows:

```
package mypkg;
public class MyLocales {
    ...
    public static boolean equals( String l1, String l2 ) {
        return l1.equals(l2);
    }
}
```

Then you map the function name as used in the EL expression to the defining class and function signature in a TLD. The following functions.tld file in the date example maps the equals function to the class containing the implementation of the function equals and the signature of the function:

```

<function>
<name>equals</name>
<function-class>mypkg.MyLocales</function-class>
<function-signature>boolean equals( java.lang.String,java.lang.String )</function-signature>
</function>

```

A tag library can have only one function element that has any given name element.

EL Functions (JSTL 1.1)

Note: All functions treat null Strings as empty Strings.

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jstl/functions" %>
```

Usage: \${fn:function(arg0, ...)}

```
<p>We offer ${fn:length(flavorSet)} ice cream flavors.</p>
```

Function	Description
fn:contains (string, substring) :boolean	Returns true if substring is contained in string; false, otherwise.
fn:containsIgnoreCase (string, substring) : boolean	Returns true if substring is contained in string regardless of case; false, otherwise.
fn:endsWith(string, suffix) :boolean	Returns true if string ends with the specified suffix; false, otherwise.
fn:escapeXml (string) : String	Escapes characters (e.g changing "<" to "<") that could be interpreted as XML (including HTML) markup.
fn:indexOf(string, substring) : int	Returns an integer representing the 0-based index within string of the first occurrence of substring. If substring is empty, 0 is returned
fn:join(string[], separator) :String	Joins all elements of the string array into a single string. Separator separates each element in the resulting string. If separator is an empty string, the elements are joined without a separator.
fn:length(collection or string) :int	If a collection or array is passed, the size of the collection or array is returned; if a string is passed, the number of characters in the string is returned.

fn:replace(inputString,beforeSubstring,afterSubstring):String	Replaces in inputString, every occurrence of beforeString with afterString. An empty string is returned if either inputString or beforeString is empty. If afterString is empty, all occurrences of the beforeString are removed.
fn:split(string, delimiters) :String[]	Splits string into a string array using the given set of delimiter characters. The delimiter characters are not included in any returned tokens.
fn:startsWith(string, prefix) :boolean	Returns true if string starts with the specified prefix; false, otherwise. Returns true if prefix is empty.
fn:substring(string, beginIndex,endIndex) : String	Returns a subset of string using the zero-based indices – inclusive of the begin index, but exclusive of the end index.
fn:substringAfter(string, substring) :String	Returns the subset of string following the given substring.
fn:substringBefore(string, substring) :String	Returns the subset of string that precedes the given substring.
fn:toLowerCase(string) : String	Converts all characters of a string to lowercase.
fn:toUpperCase(string) : String	Converts all characters of a string to uppercase.
fn:trim(string) : String	Removes whitespace from both ends of a string.

Core Tag Library

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

General-Purpose Actions

Actions for rendering data, creating and modifying scoped variables, and catching exceptions.

<c:out>: renders data to the page

```
<h2>Welcome, <c:out value="${user.name}" default="Guest"/></h2>
```

Attribute	Description	Required	Default
value	Data to output	Yes	None
default	Fallback data to output if value is empty	No	Body
escapeXml	true to escape special characters	No	true

<c:set>: saves data to a scoped variable

```
<c:set var="dogAge" value="${age div 7}" />
```

You are <c:out value="\${dogAge}" /> in dog years.

Attribute	Description	Required	Default
value	Data to save	No	Body
target	Name of variable to modify	No	None
property	Property of target to modify	No	true
var	Name of variable to store data	No	None
scope	Scope of variable	No	Page

<c:remove>: deletes a scoped variable

```
<c:remove var="dogAge" scope="page"/>
```

Attribute	Description	Required	Default
var	Name of variable to delete	yes	None
scope	Scope of variable	No	All scopes

<c:catch>: traps all exceptions or errors from the enclosed body.

```
<c:catch var="err">
<c:import value="http://java.sun.com"/>
</c:catch>
<c:if test="${not empty err}">
    Could not connect to Java web site.
</c:if>
```

Attribute	Description	Required	Default
var	Name of variable to hold the thrown exception, if any. Variable will be of type java.lang.Throwable	No	None

Conditional Actions

Actions for processing markup based on logical conditions.

```
<c:if> : processes the body if test is true
<c:if test="${user.age ge 40}">
    You are over the hill.
</c:if>
```

Attribute	Description	Required	Default
Test	Condition to evaluate	Yes	None
var	Name of variable to store test condition's result	No	None
scope	Scope of variable	No	Page

<c:choose>: multiple conditions – processes the body of the first enclosed when tag where the test condition is true. If none match then the body of the otherwise tag (if present) is processed.

```
<c:choose>
<c:when test="${a boolean expr}">
    // do something
</c:when>
<c:when test="${another boolean expr}">
```

```
// do something else

</c:when>
<c:otherwise>
// do this when nothing else is true
</c:otherwise>
</c:choose>
```

The choose tag accepts no attributes and can only contain when tag(s) and an optional otherwise tag.

<c:when>: processes the body if test is true and no other previous <c:when> tags evaluated to true.

Attribute	Description	Required	Default
test	Condition to evaluate	Yes	None

<c:otherwise>: processes the body if no other previous <c:when> condition matched. This tag accepts no attributes and, if present, must be the last tag in the <c:choose> body.

Iterator Actions

Actions that loop over collections, for a fixed number of times, or over a set of string tokens. These actions share the following attributes for iterating over a subset of elements.

Attribute	Description	Required	Default
begin	Zero-based index of first item to process, inclusive.	No	0
end	Zero-based index of last item to process, inclusive.	No	Last Item
step	Process every stepth element (e.g 2 = every second element).	No	1
varStatus	Name of variable to hold the loop status with the following properties: <ul style="list-style-type: none"> index – position of the current item count – number of times through the loop (starting with 1) first – Boolean indicator if this is the first iteration last – boolean indicator if this is the last iteration 	No	None

<c:forEach>: repeats the nested body content over a collection or for a fixed number of times.

```

<c:forEach items="\${user.languages}" var="lang" varStatus="status">
<c:if test="\${status.first}">
    You speak these languages:<br><ul>
</c:if>
<li><c:out value="\${lang}" /></li>
<c:if test="\${status.last}"></ul></c:if>
</c:forEach>

```

Attribute	Description	Required	Default
var	Name of variable to hold the current item. This variable has only nested visibility.	No	None
items	Collection, iterator, map, or array to loop over.	No	None

<c:forTokens>: repeats the nested body content for each token of a delimited string.

```

<c:set var="users">Fred,Joe,Mary<c:set>
<c:forTokens var="name" items="\${users}" delims=",">
<c:out value="\${name}" /><br/>
</c:forTokens>

```

Attribute	Description	Required	Default
var	Name of variable to hold the current token. This variable has only nested visibility.	No	None
Items	String of tokens to loop over.	Yes	None
Delims	Set of characters that separate the tokens (e.g. delims="," will tokenize a string separated by commas or semi-colons).	Yes	None

URL Related Actions

Actions for importing content from URLs, building URLs, and redirecting.

<c:import>: imports the content of a URL-based resource. Action may include nested <c:param> tags to specify the query string (unless the varReader attribute is specified).

```

<c:import url="includes/header.jsp">
<c:param name="title">Hello World</c:param>
</c:import>

```

Attribute	Description	Required	Default
url	URL of the resource to import.	Yes	None
context	Name of the context (beginning with a /) of some other local web application to import the resource from.	No	Current Context
var	Name of the variable to hold the imported content as a String.	No	None
scope	Scope of the var variable	No	Page
varReader	Name of the variable to hold the imported content as a Reader. This variable has only nested visibility so that the reader will always be closed.	No	None

<c:url>: builds a URL with the proper rewriting rules applied (only relative URLs are rewritten). Action may include nested <c:param> tags to specify the query string.

```
<c:url="editProfile.do" var="profileLnk">
<c:param name="id" value="${user.id}"/>
</c:url>
<a href='<c:out value="${profileLnk}" />'>
Edit Profile
</a>
```

Attribute	Description	Required	Default
Value	URL to be processed.	Yes	None
context	Name of the context (beginning with a /) of some other local web application.	No	Current Context
var	Name of the variable to hold the URL as a String.	No	None
scope	Scope of the var variable	No	Page

<c:redirect>: sends the client a response to redirect to the specified URL. This action will abort processing of the current page. Action may include nested <c:param> tags to specify the query string.

```
<c:if test="${empty user}">
<c:redirect url="login.do"/>
</c:if>
```

Attribute	Description	Required	Default
url	URL of the resource to redirect to.	Yes	None
context	Name of the context (beginning with a /) of some other local web application.	No	Current Context

<c:param>: adds request parameters to a URL. This action can only be nested within <c:import>, <c:url>, or <c:redirect>.

Attribute	Description	Required	Default
name	Name of the query string parameter.	Yes	None
Value	Value of the parameter. If not specified, value is taken from the tag body.	No	Body

Formatting Tag Library

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
```

Internationalization (I18N) Actions

Actions that establish localization (l10n) contexts, specify resource bundles, and format messages.

<fmt:setLocale>: Sets the default locale for the specified scope. This will override the browser-based locale.

```
<fmt:setLocale scope="session" value="fr_CA">
```

Attribute	Description	Required	Default
Value	String representation of a locale (e.g. en_US) or an actual java.util.Locale object.	Yes	None
Variant	Locale variant (as a String) to specify in conjunction with the locale (language and country).	No	None
Scope	Scope to set the default locale for.	No	Page

<fmt:bundle>: Sets the localization context, based on the specified resource bundle, to be used within the body content of this tag.

```
<fmt:bundle basename="resources" prefix="label.">
<fmt:message key="userId"/>
</fmt:bundle>
```

Attribute	Description	Required	Default
basename	Fully-qualified name of the base bundle without a file type (such as ".properties").	Yes	None
Prefix	String prefix to be prepended to the value of the message key. Note that the prefix must include <i>all</i> characters – a separator character (e.g. ".") is <i>not</i> assumed.	No	None

<fmt:setBundle>: Creates and stores in a scoped variable, a localization context based on the specified resource bundle.

```
<fmt:setBundle basename="ApplicationResources" var="strutsMessages" scope="application"/>
```

Attribute	Description	Required	Default
basename	Fully-qualified name of the base bundle without a file type (such as ".properties").	Yes	None
Var	Name of the variable to hold the localization context.	No	Default I10n Context
Scope	Scope of the var variable	No	Page

<fmt:message>: Looks up a localized message in a resource bundle. This tag can contain nested <fmt:param> tags to specify message format substitution values. The resultant message is printed or stored in a scoped variable.

```
<fmt:message key="title" bundle="${strutsResources}"/>
```

Attribute	Description	Required	Default
key	Message key to be looked up.	No	Body
bundle	Localization context (set by prio configuration, <fmt:bundle>, or <fmt:setBundle>, which specifies the resource bundle the message key is to be looked up in.)	No	Default I10n context
var	Variable to hold the message.	No	
scope	Scope of the var variable.	No	Page

<fmt:param>: Supplies a parameter for message format substitution in a containing <fmt:message> tag. Parameters are substituted in sequential order.

```
<fmt:message key="fieldRequired">
<fmt:param value="User ID"/>
</fmt:message>
```

Attribute	Description	Required	Default
value	Value used for parametric message format substitution.	No	Body

<fmt:requestEncoding>: Instructs JSTL to use a specific character encoding to decode request parameters. Omitting a value indicates to use automatic detection of the proper encoding.

```
<fmt:requestEncoding key="ISO-8859-1"/>
```

Attribute	Description	Required	Default
value	Character encoding (e.g. "UTF-8") to use.	No	Automatic

Formatting Actions

Actions that format and parse numbers, currencies, percentages, dates and times.

<fmt:timeZone>: Sets the specified time zone to be applied to the nested body content. The following example demonstrates that the time zone by this action has only nested visibility.

```
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:timeZone value="America/Los_Angeles">
Pacific Time:<fmt:formatDate type="time" timeStyle="short" value="${now}"/>
</fmt:timeZone>
<br/>
Local Time:<fmt:formatDate type="time" timeStyle="short" value="${now}"/>
```

Attribute	Description	Required	Default
value	String representation of a time zone (such as "America/New_York", "GMT-5" or "EST") or an actual java.util.TimeZone object.	Yes	None

<fmt:setTimeZone>: Sets the specified time zone in a named scoped variable or using the default time zone name if var is not specified.

```
<fmt:setTimeZone var="mtnTime" value="America/Denver"/>
```

```
Mountain Time: <fmt:formatDate type="time" timeStyle="short" value="${now}"  
timeZone="${mtnTime}"/>
```

Attribute	Description	Required	Default
Value	String representation of a time zone (such as "America/New_York", "GMT-5", or "EST") or an actual java.util.TimeZone object.	Yes	None
var	Name of the variable to store the time zone.	No	Default time zone
Scope	Scope of the var variable	No	Page

<fmt:formatNumber>: Formats a number,currency, or percentage in a locale-sensitive manner. The formatted value is printed or stored in a scoped variable.

```
<fmt:formatNumber type="currency" value="3.977">
```

Attribute	Description	Required	Default
Value	Numeric value to format.	No	Body
Type	Specifies the type of value. Valid values are: <ul style="list-style-type: none"> • number • currency • percentage 	No	Number
pattern	Custom formatting pattern (overrides other formatting options including type – see java.text.DecimalFormat)	No	none
currencyCode	Currency code (ISO 4217) used for formatting	No	Based on
	currencies. Such as "USD" (US dollars) or "EUR" (euro).		Default locale
currencySymbol	Currency symbol used when formatting currencies. Such as "\$" for US dollars, or "F" for Francs.	No	Based on Default locale
groupingUsed	Specifies if grouping separators will be used (for example – formatting "23890" as "23,890").	No	true
maxIntegerDigits	Maximum number of digits to print in the integer part of the number.	No	None
minIntegerDigits	Minimum number of digits to print in the integer part of the number.	No	None
maxFractionDigits	Maximum number of digits to print in the fractional part of the number.	No	None
minFractionDigits	Minimum number of digits to print in the fractional part of the number.	No	None
var	Variable to store the formatted number.	No	None
Scope	Scope of the var variable	No	Page

<fmt:parseNumber>: Parse a String representing a number, currency, or percentage in a locale sensitive manner.

```
<fmt:parseNumber var="num" type="number" pattern="#,###" value="2,447"/>
<c:out value="${num}" />
```

Attribute	Description	Required	Default
Value	Value to parse.	No	Body
Type	Specifies the type of value. Valid values are: <ul style="list-style-type: none"> • number • currency • percentage 	No	Number
Pattern	Custom parsing pattern (overrides type – see java.text.DecimalFormat)	No	None
parseLocale	String representation of a locale (e.g. en_US) or an actual java.util.Locale object used for parsing.	No	Default Locale
integerOnly	Specifies if only the integer portion of the value should be parsed.	No	False
var	Variable to store the formatted number.	No	None
scope	Scope of the var variable	No	Page

Formatting Dates

Dates are formatted and parsed using the <fmt:formatDate> and <fmt:parseDate> actions which share the following common attributes.

Attribute	Description	Required	Default
type	Specifies the type of value. Valid values are: <ul style="list-style-type: none"> • time (time only) • date (date only) • both (date and time) 	No	date
dateStyle	Predefined formatting style for a date (ignored if type="time") – see java.text.DateFormat. Valid values are: <ul style="list-style-type: none"> • default (Jul 19, 2003) • short (7/19/03) • medium (Jul 19, 2003) • long (July 19, 2003) • full (Saturday, July 19, 2003) 	No	default
timeStyle	Predefined formatting style for a time (ignored if type="date") – see java.text.DateFormat. Valid values are: <ul style="list-style-type: none"> • default (2:51:16 PM) • short (2:51 PM) • medium (2:51:16 PM) • long (2:51:16 PM EDT) • full (2:51:16 PM EDT) 	No	default
pattern	Custom formatting style (overrides type, dateStyle, and timeStyle) – see java.text.SimpleDateFormat.	No	None
timeZone	String representation of a time zone or an actual java.util.TimeZone object.	No	Default time zone

<fmt:formatDate>: Formats a date and/or time in a locale-sensitive manner. The formatted value is printed or stored in a scoped variable.

```
<fmt:formatDate value="${now}" pattern="yy-MMM-dd"/>
```

Attribute	Description	Required	Default
value	Date value to format. Value must be a java.util.Date object.	No	None
var	Variable to store the formatted number.	No	None
scope	Scope of the var variable	No	page

<fmt:parseDate>: Parses a string representing a date and/or time in a locale-sensitive manner. The parsed value is printed or stored in a scoped variable.

```
<fmt:parseDate var="bday" pattern="MM/dd/yy" value="05/10/63"/>
<fmt:formatDate value="${bday}" dateStyle="full"/>
```

Attribute	Description	Required	Default
value	Date/time string to parse.	No	None
parseLocale	String representation of a locale (e.g. en_US) or an actual java.util.Locale object used for parsing.	No	Default locale
var	Variable to store the parsed value as a java.util.Date.	No	None
scope	Scope of the var variable	No	Page

SQL Tag Library

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
```

The SQL tag library provides actions to perform transactional database queries and updates and easily access query results.

<sql:query>: Performs a database query (e.g. select statement). The query should be expected to return a ResultSet. This action may include body content containing the actual query string as well as <sql:param> and <sql:dateParam> tags for parameter substitution. If the body contains the SQL query, it must appear before any nested parameter tags.

```
<sql:query var="users">
  SELECT * FROM users WHERE last_name = "Burdell" </sql:query>
  <c:forEach var="user" items="${users.rows}">
    <c:out value="${user.user_name}" /><br/>
  </c:forEach>
```

Attribute	Description	Required	Default
sql	SQL query to execute.	No	Body
dataSource	Database connection to use. A javax.sql.DataSource object, or a String representing a JNDI resource or the parameters for java.sql.DriverManager. If specified, this action cannot be nested in <sql:transaction>.	No	Default data source
maxRows	Maximum number of rows to return in the result. If not specified or set to -1, no limit is enforced.	No	Default maxRows
startRow	Returned results include rows starting at this index. The first row is row 0.	No	0
var	Name of the variable to store the query results. Returned rows are accessible via the rows property of this object.	Yes	None
scope	Scope of the variable var.	No	page

<sql:update>: Performs a database insert, update, delete or other statement (such as a DDL statement) that does not return any results. This action may include body content containing the actual update string as well as <sql:param> tags for parameter substitution. Like <sql:query>, the <sql:param> tags must occur after the sql statement if it is contained in the <sql:update> tag body.

```
<sql:update var="updateCount">
    UPDATE users SET first_name="William" WHERE first_name = "Bill"
</sql:update>
<c:out value="#{updateCount} rows updated."/>
```

Attribute	Description	Required	Default
sql	SQL statement to execute.	No	Body
dataSource	Database connection to use. A javax.sql.DataSource object, or a String representing a JNDI resource or the parameters for java.sql.DriverManager. If specified, this action cannot be nested in <sql:transaction>.	No	Default Data Source
var	Name of the variable to store the query results as a java.lang.Integer object (e.g. SQL update statements return the number of rows affected).	No	None
scope	Scope of the variable var.	No	page

<sql:transaction>: Establishes a database transaction for <sql:query> and <sql:update> tags contained in this tag's body. That is, all SQL actions contained in the body of this tag will be treated as a single atomic transaction. The transaction will be committed only if all statements succeed. If any of the contained SQL actions cause an exception, the transaction will be rolled back.

Note: This action will reinstate the prior “auto commit” setting after completion.

```
<sql:transaction>
<sql:update sql="DELETE users WHERE user_name='bsiggelkow'" />
<sql:update sql="INSERT INTO users VALUES ('billsigg','Bill','Siggy')"/>
</sql:transaction>
```

Attribute	Description	Required	Default
dataSource	Database connection to use. A javax.sql.DataSource object, or a String representing a JNDI resource or the parameters for java.sql.DriverManager.	No	Default data Source
isolation	Transaction isolation level. Valid values are: <ul style="list-style-type: none">• read_committed• read_uncommitted• repeatable_read• serializable	No	Database's Default

<sql:setDataSource>: Creates and stores in a scoped variable an SQL data source. This tag cannot have a body. Either the dataSource or url attribute must be specified.

```
<sql:setDataSource var="testDB" url="jdbc:mysql://localhost/test,com.mysql.jdbc.Driver"/>
<sql:query var="users" dataSource="${testDB}">
    SELECT * FROM users
</sql:query>
```

Attribute	Description	Required	Default
dataSource	Database connection to use or create. A javax.sql.DataSource object, or a String representing a JNDI resource or the parameters for java.sql.DriverManager.	No	None
driver	Name of the JDBC driver class.	No	None
url	URL for the JDBC database connection.	No	None
user	Database user's username	No	None
password	Database user's password	No	None
var	Name of the variable to hold the data source.	No	Default data Source name
scope	Scope of the variable var.	No	Page

<sql:param>: Substitutes parameters into SQL prepared statement placeholders ("?"). This action can only be nested within <sql:query> or <sql:update> tags. If the enclosing tag specifies the SQL in the body, any <sql:param> tags must appear after the SQL.

Parameters are substituted in sequential order.

```
<c:set var="firstName">Bill</c:set>
<sql:query var="users">
    SELECT user_name, last_name FROM users WHERE first_name = ?
    <sql:param value="${firstName}" />
</sql:query>
```

Attribute	Description	Required	Default
value	Value of the parameter. If not specified, value is taken from the tag body.	No	Body

<sql:dateParam>: Substitutes time, date, or timestamp parameters into SQL prepared statement placeholders. This action can only be nested within `<sql:query>` or `<sql:update>` tags. Parameters are substituted in sequential order.

```

<fmt:parseDate var="ww2End" pattern="yyyy-MM-dd" value="1945-09-02"/>
<sql:query var="postWarBabies">
    SELECT user_name FROM user_profile WHERE birth_date > ?
    <sql:dateParam value="${ww2End}" type="date"/>
</sql:query>

```

Attribute	Description	Required	Default
value	Value of the parameter. Must be a <code>java.util.Date</code> object.	Yes	None
type	Indicates how the database should interpret the value. Valid values are: <ul style="list-style-type: none"> • date • time • timestamp 	No	timestamp

XML Tag Library

```
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>
```

The XML tag library supports parsing of XML documents, selection of XML fragments, conditional and iterative processing based on XML content, and XSLT transformations. A common pattern for using the XML tags is as follows:

1. Use `<x:parse>` to parse XML into a scoped variable. The XML can come from the body literally, from the body via `<c:import>`, or from the value attribute which may refer to any XML source.

```
<x:parse var="varName" ...>
```

2. Use the scoped variable from `<x:parse>` to specify the XML document to use in XPath expressions.

```
<x:out select="$varName/XPathexpression"/>
```

Using JSTL Data as XPath Variables

Scoped variables can be used in XPath expressions (`$implicitObject:variableName`) similar to how they are used in EL (`${implicitObject.variableName}`). If the implicit object is omitted, scopes will be searched in standard order.

Note that the “.” and “[]” notations cannot be used for accessing bean properties.

The following example demonstrates the above techniques.

```
<%-- Create the XML --%>
<x:parse var="doc">
    <users>
        <user id="997">
            <first-name>George</first-name>
            <last-name>Burdell</last-name>
        </user>

        <user id="998">
            <first-name>Joseph</first-name>
            <last-name>Blough</last-name>
        </user>
    </users>
</x:parse>
<%-- Define a variable holding user ID --%>
<c:set var="userId" value="${user.id}" scope="page"/>
<%-- Find the user with matching ID --%>
<x:set var="user" select="$doc//users/user[@id=$pageScope:userId]"/>
<%-- Say Hi to the user --%>
Hi <x:out select="$user/first-name"/> !
```

XPath Abbreviated Syntax

Abbr.	XPath Axis	Example
//	Descendant	\$doc//first-name
.	Self	\$doc/users/user/.
..	parent	\$doc//[firstname='George']/..lastname
@	attribute	\$doc//user[@id="997"]/first-name
(blank)	child	\$doc/users/user[lastname='Burdell']
*	(matches any namespace)	\$doc//*[@localname()='users']/user

XPath Functions

Following are some useful XPath functions that can be used in XPath expressions. This list does not include all XPath functions – just those deemed particularly useful when using the JSTL XML tags.

Function	Description
<code>ceiling(number or object):number</code>	Returns the smallest integer greater than or equal to <i>number</i> or <i>object</i> (as converted as if by the <code>number()</code> function).
<code>concat(string1, string 2,...):string</code>	Concatenates the arguments in order from left to right. Non-string arguments are converted to strings as if by the <code>string()</code> function.
<code>contains(string1, string2):Boolean</code>	Returns true if <i>string1</i> contains <i>string2</i> ; false, otherwise. Non-string arguments are converted to strings as if by the <code>string()</code> function.
<code>count(node-set):number</code>	Returns the number of nodes in the node set.
<code>floor(number or object):number</code>	Returns the greatest integer less than or equal to <i>number</i> or <i>object</i> (as converted as if by the <code>number()</code> function).
<code>last():number</code>	Returns the number of nodes in the context node.
<code>local-name(node-set):string</code>	Returns the part of the element name <i>after</i> the colon (:) when namespaces are used. If node set is not specified the function is applied to the context node.
<code>name(node-set):string</code>	Returns the qualified element name, that is, <i>prefix:local-name</i> , when namespaces are used. If node set is not specified the function is applied to the context node.
<code>namespace-uri(node-set):string</code>	Returns the uri of the namespace of the given node set. If node set is not specified the function is applied to the context node.
<code>not(boolean or object):boolean</code>	Returns the logical inversion of the supplied argument. That is, if the argument is true, false is returned; if argument is false, true is returned.
<code>number(object):number</code>	Converts object (or the current node if object is not specified) to a number. True is converted to 1, and false to 0. If the number cannot be converted NaN is returned.
<code>position():number</code>	Returns the position of the current node in the current context node set. The first position is 1.
<code>round(number or object):number</code>	Returns the closest integer to number or object (as converted as if by the <code>number()</code> function). If two integers are equally close, the value closer to positive infinity is chosen. (e.g. <code>round(3.5)</code> returns 4, <code>round(-3.5)</code> returns -3).
<code>starts-with(string1,string2):boolean</code>	Returns true if <i>string1</i> starts with <i>string2</i> ; false, otherwise.
<code>string(object):string</code>	Converts the object, or the current node if object is not specified, to a string. Generally, this function will output the body of the elements (e.g. <code>string(<a>b)</code> returns "b").
<code>string-length(object):number</code>	Number of characters in the string. If object is not a string it is first converted to a string as if by the <code>string()</code> function.
<code>substring(string, index,length):string</code>	Returns a substring of string starting at index and continuing for length characters. The first character is position 1, not position 0 as in Java and JavaScript.
<code>substring-after(string1, string2):string</code>	Returns the substring in <i>string1</i> following the first occurrence of <i>string2</i> in <i>string1</i> .
<code>substring-before(string1, string2):string</code>	Returns the substring in <i>string1</i> preceding the first occurrence of <i>string2</i> in <i>string1</i> .
<code>sum(node-set):number</code>	Converts each node in the set to a number, as if by the <code>number()</code> function, adds up the numbers and returns the sum.

General XML Actions

Actions for parsing XML, outputting to the page, and selecting XML fragments. The examples that follow demonstrate use of the XML tags for processing Rich Site Summary (RSS) feeds. RSS has more or less the following format:

```
<?xml version="1.0"?>
<rss version="0.91">
<channel>
<title>feed title</title>
<link>source url</link>
<description>feed desc</description>
<item>
<title>article title</title>
<link>article url</link>
<description>articledesc</description>
</item>
<item>

<title>article title</title>
<link>article url</link>
<description>articledesc</description>
</item>
</channel>
</rss>
```

<x:parse>: Parses XML content, provided by the value attribute or the tags body, into a scoped variable(s). This variable can then be used for subsequent processing by other XML tags.

```
<c:import      var="rss"      url="http://servlet.java.sun.com/syndication/rss_java_highlights-PARTNER-20.xml"/>

<x:parse var="news" xml="${rss}"/>
```

Attribute	Description	Required	Default
doc	(JSTL 1.1) Text of the document to a parse as a String or Reader.	No	Body
xml	(JSTL 1.0 – deprecated in 1.1) Same as doc attribute.	No	Body
systemId	URI of the original document, used for entity resolution.	No	None
Filter	XML filter to apply. Value should be of type org.xml.sax.XMLFilter.	No	None
var	Name of the variable to store the results – may be an implementation-specific object.	Yes	None
scope	Scope of the variable var.	No	page
varDom	Name of the variable to store the result – stored as a DOM object.	Yes	None
scopeDom	Scope to store variable varDOM in.	No	Page

<x:out>: Prints the result of the XPath expression as a string. RSS Channel:

```
<x:out select="$news//channel/title"/>
```

Attribute	Description	Required	Default
Select	XPath expression.	Yes	None
escapeXml	true to escape special characters	No	True

<x:set>: Saves the result of the select XPath expression to a scoped variable. Returned value may be a node set (XML fragment), boolean, string, or number.

```
<x:set select="$news//channel/item" var="newsItems"/>
```

Attribute	Description	Required	Default
Select	XPath expression.	No	None
var	Name of variable to store results. The variable type is equal to whatever is returned by the select expression. <ul style="list-style-type: none"> • Boolean – java.lang.Boolean • Number – java.lang.Number • Node or node set – Implementation dependent type 	No	None
scope	Scope of variable var.	No	page

Flow Control Actions

Actions for processing markup based on logical and iterative conditions.

<x:if>: Processes the body if select XPath evaluates to true (following the rules of the boolean() XPath function).

```
<x:if select="$news//item[contains(description,'Linux')]">
    Linux is in the news today!
</x:if>
```

Attribute	Description	Required	Default
select	Test condition as an XPath expression. Body content is processed if the condition is true.	Yes	None
var	Name of variable to store test condition's result	No	None
scope	Scope of variable var.	No	page

<x:choose>: Processes the body of the first enclosed <x:when> tag where the select XPath expression evaluates to true. If none match then the body of the <x:otherwise> tag (if present) is processed.

```
<x:choose>
<x:when select="$news//item">
    We've got news :)
```

```

</x:when>
<x:otherwise>
No news today :(
</x:otherwise>
</x:choose>

```

The choose tag accepts no attributes and can only contain <x:when> tag(s) and an optional <x:otherwise> tag.

<x:when>: Represents an alternative in an <x:choose> tag. Processes the body if the select expression evaluates to true and no other previous <x:when> tags in the <x:choose> matched.

Attribute	Description	Required	Default
select	Test condition as an XPath expression.	Yes	None

<x:otherwise>: Processes the body if no other previous <x:when> condition in the <x:choose> matched. This tag accepts no attributes and, if present, must be the last tag in <x:choose> body.

<x:forEach>: Repeats the nested body content over a node set determined by an XPath expression, setting the context node to each element in the iteration.

```

<x:forEach select="$news//item">
<a href='<x:out select="link"/>'>
<x:out select="title"/></a><br/>
</x:forEach>

```

Attribute	Description	Required	Default
select	Name of variable to hold the current item. This variable has only nested visibility.	No	None
var	Name of variable to hold the current item. This variable has only nested visibility.	No	None

Transformation Actions

JSTL provides an <x:transform> tag for performing XSLT transformations. The <x:param> tag can be nested in the <x:transform> tag to set a parameter that is used in the stylesheet.

<x:transform>: Conducts an XSLT transformation on source XML. The source XML is provided by the doc attribute or the body of the tag. The XSL stylesheet is specified by the xslt attribute. While in most cases, the stylesheet will be set up by back-end code – it is possible to define the stylesheet inline and make it available with <c:set> as in the following example:

```

<c:set var="xsl">
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="item">

```

```

<li>
<xsl:value-of select="title"/>
</li>
</xsl:template>
<xsl:template match="/">
<ol>
<xsl:apply-templates select="//item"/>
</ol>
</xsl:template>
</xsl:stylesheet>
</c:set>
<x:transform xml="${news}" xslt="${xsl}"/>

```

Attribute	Description	Required	Default
doc	(JSTL 1.1) Source XML document for the transformation. Value can be a String, Reader, javax.xml.transform.Source , org.w3c.dom.Document, or any value exported <x:parse> or <x:set>. If the value comes from <x:set> it cannot be a partial document.	No	Body
xml	(JSTL 1.0 – deprecated in 1.1) Same as doc attribute.	No	Body
docSystemId	(JSTL 1.1) URI of the source XML to used to resolve entity references.	No	None
xmlSystemId	(JSTL 1.0 – deprecated in 1.1) Same as docSystemId.	No	None
xslt	XSLT stylesheet to use. The value must be a String, Reader or an javax.xml.transform.Source object.	yes	none
xsltSystemId	URI of the stylesheet to use to resolve entity references.	no	none
var	Variable to hold the result as a org.w3c.doc.Document object.	no	None
scope	Scope of the variable var.		
result	javax.xml.transform.Result object that holds or processes the transformed XML.	No	None

<x:param>: Sets a transformation parameter that will be passed to stylesheet which declares parameters using the <xsl:param> tag. The <x:param> tag can only be nested within an <x:transform> tag. Any <x:param> tags must come after the XML body content of the <x:transform> tag, if present.

```

<x:transform xml="${news}" xslt="${searchXsl}"/>
<x:param name="searchParam">
    Web Services
</x:param>
</x:transform>

```

Attribute	Description	Required	Default
name	Name of the parameter as a String. This name must match the name in the corresponding <xsl:param> XSLT tag.	Yes	None
value	Value of the parameter. If not specified, value is taken from the tag body.	No	Body

MVC Based Web Development with Struts

Struts

Apache Struts is an open source Java framework used for building web applications based on the servlet and JavaServer Pages (JSP) technologies. It was created in 2000 by Craig R. McClanahan and has become the de facto standard framework for web applications in Java.

The Web Development Landscape

This is going to sound like a grandfather preaching to his grandkids about how tough it was in his day, but prior to 2000, the landscape for developing Java web applications was much different. Java servlets had been out for just a short time (relatively speaking) and Sun had recently released JavaServer Pages (JSP) to counter Microsoft's Active Server Pages (ASP) technology. JSPs caught on like wildfire and before long, many web applications were being built using JSPs.

Unfortunately, in an effort to get applications quickly out the door and due to a lack of any best practices in the industry at the time, many JSP applications became a tangled web of pages as developers crammed as much of the application functionality into the JSP. This approach to building JSP applications would eventually be called the JSP "Model 1" architecture and is shown in Figure 1.

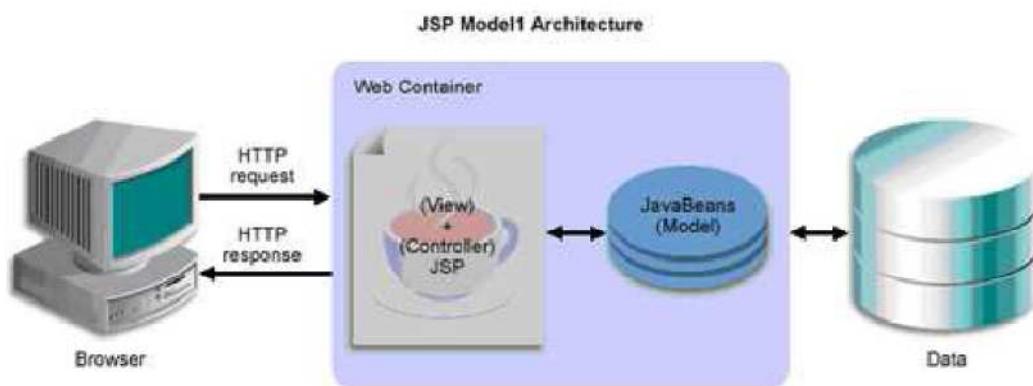


Figure 1. The JSP Model 1 architecture

The Model 1 architecture used the JSPs to handle the HTTP requests, communicate with the data layer (via JavaBeans), and to render the next page back to the client. For small applications, this approach isn't so terrible. But for non-trivial applications where there are many pages or when the

functionality is more traditionally application-like, the Model 1 architecture presents many maintenance and upgradeability challenges.

Developers soon realized that the use of JSPs in conjunction with a single controller servlet could improve this design greatly. The separation of the presentation layer from the business logic not only serves to keep the JSPs cleaner, but it allows development teams to utilize each team member's specialized skill.

This new approach to building JSP applications became known as the JSP "Model 2" architecture. It was patterned after the Model-View-Controller (MVC) design, which originated from Smalltalk, where it was used to separate the GUI from the business layer. (When MVC is done on the Web, it is often referred as Web MVC.) The JSP Model 2 architecture is shown in Figure 2.

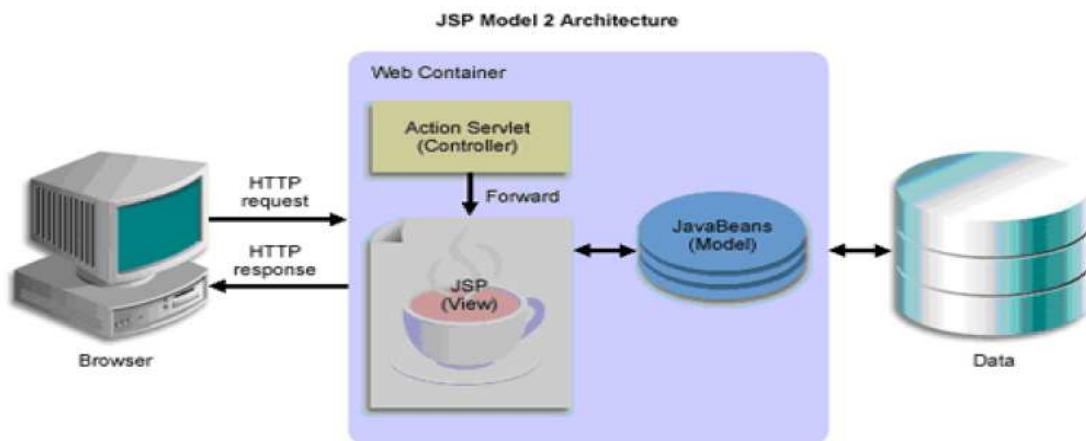


Figure 2. The JSP Model 2 architecture

The Model 2 architecture uses a Java servlet to receive the HTTP requests from the browser and to communicate with the model layer. The JSPs are used only for presentation. The same controller that receives the request is responsible for determining which view gets returned back to the client.

Along Comes Jakarta Struts

Three things started to happen around 2000 that helped catapult Struts into the forefront of web development frameworks. First, the open source community was starting to gain momentum and popularity. Open source projects like Tomcat, Ant, and Log4J were gaining mindshare with the development community, and managers were just starting to accept that free software had a place in the business world.

The second thing that was happening around the same time was a huge number of new startup companies (affectionately called dot coms) were trying out their various business models on the populace. Most of these business models involved some form of web presence and therefore required a web framework. (In retrospect, for those of us who lived through it, these times were very stressful and exciting at the same time.)

It was also during this time that Craig McClanahan created the open source web development framework called Struts, which was based on the Model 2 architecture and which was added to the list of Jakarta projects. Little did he probably know that the planets were lining up behind his newly created web framework.

Benefits of the Struts Framework

There are literally hundreds of web development frameworks available, each sharing some similarities while providing their own twist on what a web framework should be. Many of these frameworks have been around several years and have positives and negatives associated with them. While the Struts framework is not completely unique, it provides some key benefits:

- Based on a Model 2 MVC implementation.
- Supports Java's "Write Once, Run Anywhere" philosophy.
- Supports different model implementations (JavaBeans, EJB, etc.).
- Support for internationalization (I18N).
- Includes a rich set of JSP tag libraries.
- Contains many extension points for customization.
- Supports different presentation implementations (JSP, XML/XSLT, JavaServer Faces).

Following is a list of reasons that would help developers decide if Struts was right for them.

- Developed by industry experts
- Stable and mature
- Open source (No cost)
- Large user community (several thousand)
- It's probably similar to what you would build if not using Struts
- Feature-rich
- Free to develop and deploy
- Many supported third-party tools
- Flexible and extendable
- J2EE technologies

The above list shows that Struts is a serious framework. After all, do you think a company the size of BEA would choose Struts for their commercially released administrative console if they didn't have confidence in the open source framework?

Overview of the Struts Framework

The Struts framework is composed of approximately 300 classes and interfaces organized in about 12 top-level packages. Besides the utility and helper classes, much of the framework consists of classes and interfaces for working with either the controller functionality or the presentation through custom tag libraries. The choice of a model layer is a decision that's left up to you. Some argue that this is a deficiency in the framework, while seasoned Struts developers would say that

this freedom is warranted and welcomed. A high-level view of the Struts architecture is shown in Figure 3.

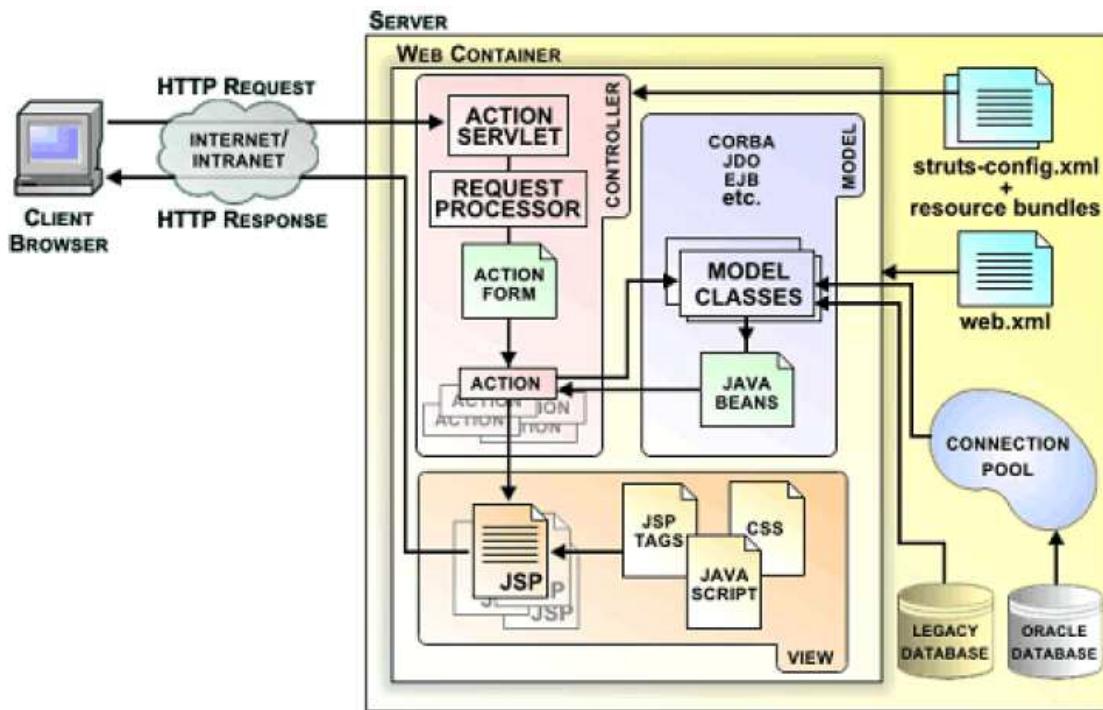


Figure 3. High Level View of Struts Architecture

The Struts Controller Components

When a request is sent to a Struts application, it's handled by the Struts ActionServlet. The Struts framework includes a concrete ActionServlet that for many users is adequate and requires no customization or additional work.

When the ActionServlet receives a request, it inspects the URL and based on the Struts configuration files, it delegates the handling of the request to an Action class. As you can see from Figure 3, the Action class is part of the controller and is responsible for communicating with the model layer. The Struts framework provides an abstract Action class that you must extend for your own needs.

Following is an example of an Action class from a typical Struts-based web application that extends the base Action class.

Example: 12.1

```
public class LoginAction extends Action {
    /**
     * Called by the ActionServlet when the a user attempts to login. */
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
```

```

/*
 * Get the user's login name and password. They should have been
 * validated by the ActionForm.
 */
String username = ((LoginForm) form).getUsername();
String password = ((LoginForm) form).getPassword();
// Login through the security service
CartService serviceImpl = this.allService();
// Authenticate the user with the backend
UserView userView = serviceImpl.authenticate(username, password);
// Invalidate the current session and create a new one
HttpSession session = request.getSession(false);
session.invalidate();
session = request.getSession(true);
// Store the user object into the session
session.setAttribute("USER", userView);

return mapping.findForward("success");
}
}

```

The Action class in the above example is called when a user attempts to log in to the website. The URL that the browser would have sent to the ActionServlet would have looked something like `http://localhost:8080/myStrutsApp/login.do`. When the ActionServlet gets this request, it consults its mapping (which was loaded from the Struts configuration files at startup) and discovers that the `LoginAction` is the Action class that handles the `login.do` request. The ActionServlet then calls the Action's `execute()` method to perform whatever task the Action class was meant to perform. Within the `execute()` method, you can put in whatever logic you need in order to fulfill the request. For example, you could acquire a Hibernate session and query the database, invoke a Stateless Session Bean (SLSB) or maybe use JavaMail to send an email to a registered user, just to name a few.

The `LoginAction` in the above example authenticates the user through some service presumably to a database. When the Action class is finished performing its logic, it typically stores objects (JavaBeans) into the request, session, or even application scope, and informs the controller servlet whether it was successful or not. Based on the mappings for the `LoginAction` class, the ActionServlet will determine the next page the user should be shown. Because the JSP has access to the same request, session, or application scope, it can access those stored objects and use them to render the view for the client.

The Struts framework includes several pre-built Actions that make it easier to build out functionality. You don't always have to map a web request to a separate Action class. The `org.apache.struts.actions.DispatchAction`, for example, is designed to support multiple requests. In a

typical e-commerce shopping cart application, for example, functionalities might be view the cart, add items, checkout, etc. Instead of creating separate Action classes for each action, we can use the DispatchAction to contain all of the methods that work on the user's cart. Using the DispatchAction class keeps the number of Action classes to a minimum, which helps to simplify the maintenance of the application.

Choosing a Model Layer for Struts

As we said earlier, the Struts framework doesn't provide much in the way of the model, but it can integrate with any current model approach, including JDBC and EJB, as well as third-party packages like Hibernate, iBATIS, and Object Relational Bridge. One of the most important best practices for the Struts framework is that business logic should stay out of the Action class, regardless of what you're using for the Model layer. Putting business logic in the Action classes hurts reusability of the Action class and couples the framework too tightly with the business tier. Instead, the Action classes should utilize a service layer, even when using straight JDBC. Back in the Example 12.1, for example, the authenticate() method was called on a service called CartService. This might have been a Stateless Session Bean, a RMI object, or maybe just a simple Java class that wraps JDBC code. The good news is that you're not limited with Struts.

The Struts Presentation Layer

The Struts presentation layer is almost as flexible as the model layer. You can (and most users do) use JavaServer Pages, of course. But you can also use one of many alternative presentation technologies if JSP isn't your thing. From Velocity Templates to XSLT to JavaServer Faces, Struts easily integrates with many.

If you are using JSP, then you're in luck, because Struts has a fairly exhaustive set of JSP custom tags.

The Venerable Struts Tag Library

The support for custom tags was added to the JSP language to allow developers to extend the JSP library. The Struts framework includes over 60 JSP custom tags grouped into one of the five libraries shown in Table 12.1.

Table 12.1. Struts includes many custom tags that make building JSP applications easier.

Tag Library	Purpose
HTML Tag Library	Contains tags used to create Struts input forms, as well as other tags generally useful in the creation of HTML-based user interfaces.
Bean Tag Library	Contains tags useful for accessing beans and their properties, as well as defining new beans.
Logic Tag Library	Contains tags that are useful in managing conditional generation of output text, looping over object collections for repetitive generation of output text, and application flow management.
Nested Tag Library	Contains tags useful for dealing with object graphs and nested properties.
Tiles Tag Library	Tiles built on the "include" feature provided by the JavaServer Pages specification to provide a full-featured, robust framework for assembling presentation pages from component parts.

The Struts framework also works quite nicely with the JSP Standard Tag Library (JSTL), which was created to provide a set of standard tags that will work in all compliant JSP containers.

Presentation Validation

Another great feature of Struts is the presentation validation that's included with the Commons Validator. The Validator framework was created by David Winterfeldt and integrated into Struts in version 1.1. The Validator provides for presentation validation (and with some customization, business validation) by declaring the rules in an external XML file. The framework comes with many pre-built validation rules, like checking for required fields, max and min lengths, and date validations. It also supports regular expressions. You can even create your own validation rules and easily add them to your application.

Struts' Other Great Features

- Support for modules
- Declarative exception-handling
- Dynamic ActionForms
- New config package
- Tiles
- Plugins

How Struts Works

The basic purpose of the Java Servlets in struts is to handle requests made by the client or by web browsers. JavaServerPages (JSP) are used to design the dynamic web pages, Servlets help to route requests which are made by the web browsers to the appropriate ServerPage. The use of servlet as a router helps to make the web applications easier to design, create, and maintain. Struts is purely based on the Model- View- Controller (MVC) design pattern. It is one of the best and most well

developed design patterns in use. By using the MVC architecture we break the processing in three sections named Model, the View, and the Controller. Below we are describing the working of struts.

1. As we all are well aware of the fact that each application we develop has a deployment descriptor i.e. WEB-INF/web.xml. This is the file which the container reads.

This file has all the configuration information which we have defined for our web application. The configuration information includes the index file, the default welcome page, the mapping of our servlets including path and the extension name, any init parameters, information related to the context elements.

In the file WEB-INF/web.xml of struts application we need to configure the Struts ActionServlet which handles all the request made by the web browsers to a given mapping. ActionServlet is the central component of the Struts controller. This servlet extends the HttpServlet.

This servlet basically performs two important things.

A) When the container gets start, it reads the Struts Configuration files and loads it into memory in the init() method. You will know more about the Struts Configuration files below.

B) It intercepts the HTTP request in the doGet() and doPost() method and handles it appropriately.

2. In struts application we have another xml file which is a Struts configuration file named as struts.config.xml. The name of this file can be changed. The name of the struts configuration file can be configured in the web.xml file. This file is placed under the WEB-INF directory of the web application. It is an XML document that describes all or part of Struts application. This file has all the information about many types of Struts resources and configures their interaction. This file is used to associate paths with the controller components of your application., known as Action classes like <action path ="/login" type = "LoginAction">. This tag tells the Struts ActionServlet that whenever the incoming request is http://myhost/myapp/login.do, then it must invoke the controller component LoginAction. Above, you can see that we have written .do in the URL. This mapping is done to tell the web application that whenever a request is received with the .do extension then it should be appended to the URL.

3. For each action we also have to configure Struts with the names of the resulting pages that will be shown as a result of that action. In our application there can be more than one view which depends on the result of an action. One can be for a success and the other for the failure. If the result action is "success" then the action tells the ActionServlet that the action has been successfully accomplished or vice- versa. Struts knows how to forward the specific page to the concerned destination. The model which we want to use is entirely to you, the model is called from within the controller components.

4. Action can also get associated with a JavaBean in our Struts configuration file. Java bean is nothing but a class having getter and setter methods that can be used to communicate between the

view and the controller layer. These java beans are validated by invoking the validate() method on the ActionForm by the help of the Struts system. The client sends the request by the normal form submission by using Get or Post method, and the Struts system updates that data in the Bean before calling the controller components.

5. The view we use in the struts can be either Jsp page, Velocity templates, XSLT pages etc. In struts there are set of JSP tags which has been bundled with the struts distribution, but it is not mandatory to use only Jsp tags, even plain HTML files can be used within our Struts application but the disadvantage of using the html is that it can't take the full advantage of all the dynamic features provided in the struts framework.

The framework includes a set of custom tag libraries that facilitate in creating the user interfaces that can interact gracefully with ActionForm beans. The struts Jsp taglibs has a number of generic and struts specific tags which help you to use dynamic data in your view. These tags help us to interact with the controller without writing much java code in the jsp. These tags are used create forms, internally forward to other pages by interacting with the bean and help us to invoke other actions of the web application.

There are many tags provided to you in the struts frameworks which helps you in sending error messages, internationalization etc.

Note: The points we have described above will be in effect if and only if when the ActionServlet is handling the request. When the request is submitted to the container which call the ActionServlet, make sure that the extension of the file which we want to access should have the extension .do.

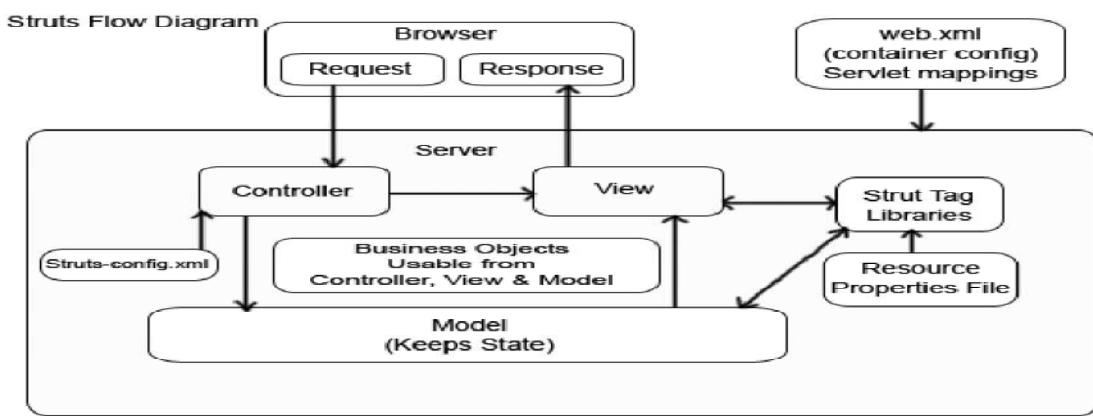


Figure 4. Working of a Struts Application

Process flow:

web.xml : Whenever the container is started up, the first work it does, is to check the web.xml file and determine what struts action servlets exist. The container is responsible for mapping all the file request to the correct action Servlet.

A Request : This is the second step performed by the container after checking the web.xml file. In this, the user submits a form within a browser and the request is intercepted by the controller.

The Controller : This is the heart of the container. Most Struts application will have only one controller that is ActionServlet which is responsible for directing several Actions. The controller determines what action is required and sends the information to be processed by an action Bean. The key advantage of having a controller is its ability to control the flow of logic through the highly controlled, centralized points.

struts.config.xml : Struts has a configuration file to store mappings of actions. By using this file there is no need to hard code the module which will be called within a component. The one more responsibility of the controller is to check the struts.config.xml file to determine which module to be called upon an action request. Struts only reads the struts.config.xml file upon start up.

Model : The model is basically a business logic part. This is a great place to perform the preprocessing of the data received from request. It is possible to reuse the same model for many page requests. Struts provides the ActionForm and the Action classes which can be extended to create the model objects.

View : The view in struts framework is mainly a jsp page which is responsible for producing the output to the user.

Struts tag libraries : These are struts components that help us to integrate the struts framework within the project's logic. These struts tag libraries are used within the JSP page. This means that the controller and the model part can't make use of the tag library but instead use the struts class library for struts process control.

Properties file : It is used to store the messages that an object or page can use. Properties files can be used to store the titles and other string data. We can create many properties files to handle different languages.

Business objects : It is the place where the rules of the actual project exists. These are the modules which just regulate the day- to- day site activities.

The Response : This is the output of the View JSP object.

Understanding Struts Controller

In this section we will describe the Controller part of the Struts Framework. We will see how to configure the struts-config.xml file to map the request to some destination servlet or jsp file.

The class org.apache.struts.action.ActionServlet is the heart of the Struts Framework. It is the Controller part of the Struts Framework. ActionServlet is configured as Servlet in the web.xml file as shown in the following code snippets.

```

<!-- Standard Action Servlet Configuration (with debugging) -->
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

```

This servlet is responsible for handling all the request for the Struts Framework, user can map the specific pattern of request to the ActionServlet. `<servlet-mapping>` tag in the web.xml file specifies the url pattern to be handled by the servlet. By default it is *.do, but it can be changed to anything. Following code from the web.xml file shows the mapping.

```

<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

The above mapping maps all the requests ending with .do to the ActionServlet. ActionServlet uses the configuration defined in struts-config.xml file to decide the destination of the request. Action Mapping Definitions (described below) is used to map any action. For this section we will create Welcome.jsp file and map the "Welcome.do" request to this page.

Welcome.jsp

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %> <html:html locale="true">
<head>
    <title><bean:message key="welcome.title"/></title>
</head>
    <body bgcolor="white">
        <h3><bean:message key="welcome.heading"/></h3>
        <p><bean:message key="welcome.message"/></p>
    </body>
</html:html>

```

Forwarding the Welcome.do request to Welcome.jsp

The "Action Mapping Definitions" is the most important part in the struts-config.xml. This section takes a form defined in the "Form Bean Definitions" section and maps it to an action class.

Following code under the <action-mappings> tag is used to forward the request to the Welcome.jsp.

```
<action path="/Welcome" forward="/pages/Welcome.jsp"/>
```

To call this Welcome.jsp file we will use the following code.

```
<html:link page="/Welcome.do">First Request to the controller</html:link>
```

Once the user clicks on 'First Request to the controller' link on the index page, request (for Welcome.do) is sent to the Controller and the controller forwards the request to Welcome.jsp. The content of Welcome.jsp is displayed to the user.

Understanding Struts Action Class

In this section, we will see how to use Struts Action Class and forward a jsp file through it.

What is Action Class?

An Action class in the struts application extends Struts "org.apache.struts.action.Action" Class. Action class acts as wrapper around the business logic and provides an interface to the application's Model layer. It acts as glue between the View and Model layer. It also transfers the data from the view layer to the specific business process layer and finally returns the processed data from business layer to the view layer.

An Action works as an adapter between the contents of an incoming HTTP request and the business logic that corresponds to it. Then the struts controller (ActionServlet) selects an appropriate Action and creates an instance if necessary, and finally calls execute method.

To use the Action, we need to subclass and overwrite the execute() method. In the Action class don't add the business process logic, instead move the database and business process logic to the process or DAO layer.

The ActionServlet (command) passes the parameterized class to Action Form using the execute() method. The return type of the execute method is ActionForward which is used by the Struts Framework to forward the request to the file as per the value of the returned ActionForward object.

Developing our Action Class?

Our Action class (TestAction.java) is simple class that only forwards to TestAction.jsp. Our Action class returns the ActionForward called "testAction", which is defined in the struts-config.xml file (action mapping is shown later). Here is code of our Action Class:

TestAction.java

```
package hpes.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
public class TestAction extends Action
{
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception{
        return mapping.findForward("testAction");
    }
}
```

Understanding Action Class

Here is the signature of execute method of the Action class.

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response) throws java.lang.Exception
```

Action class process the specified HTTP request, and create the corresponding HTTP response (or forward to another web component that will create it), with provision for handling exceptions thrown by the business logic. Returns an ActionForward instance describing where and how control should be forwarded, or null if the response has already been completed.

Parameters:

mapping:	The ActionMapping used to select this instance
form:	The optional ActionForm bean for this request (if any)
request:	The HTTP request we are processing
response:	The HTTP response we are creating
Throws:	Action class throws java.lang.Exception - if the application business logic throws an exception.

Adding the Action Mapping in the struts-config.xml

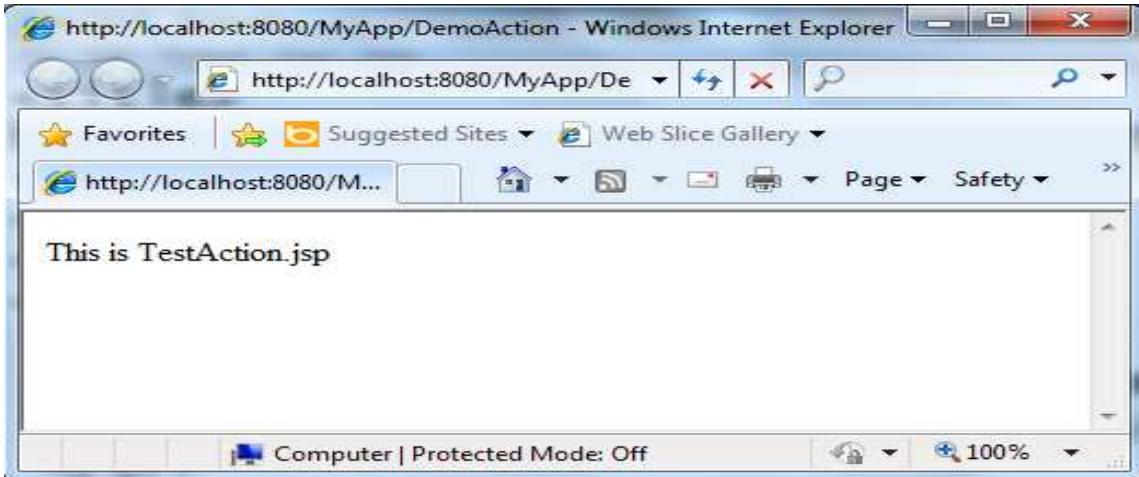
To test the application we will add a link in the index.jsp

```
<html:link page="/TestAction.do">Test the Action</html:link>
```

Following code under the <action-mappings> tag is used to for mapping the TestAction class.

```
<action path="/TestAction" type="hpes.actions.TestAction">
    <forward name="testAction" path="/pages/TestAction.jsp"/>
</action>
```

To test the new application click on Test the Action link on the index page. The content of TestAction.jsp should be displayed on the user browser.



In this section, you learned how to create Action Class and add the mappings in the struts-config.xml. Our Action Class returns the ActionForward mapping of the TestAction.jsp.

The Struts ActionForm Class

In this section, you will learn about the ActionForm in detail. We will see a good example of ActionForm. This example will help you understand Struts in detail. We will create user interface to accept the address details and then validate the details on server side. On the successful validation of data, the data will be sent to model (the action class). In the Action class we can add the business processing logic but in this case we are just forwarding it to the sucess.jsp.

What is ActionForm?

An ActionForm is a JavaBean that extends org.apache.struts.action.ActionForm. ActionForm maintains the session state for web application and the ActionForm object is automatically populated on the server side with data entered from a form on the client side.

We will first create the class AddressForm which extends the ActionForm class. Here is the code of the class:

AddressForm.java

```
package hpes.beans;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;
/***
 * Form bean for the Address Entry Screen.
 *
 */
public class AddressForm extends ActionForm {
    private String name=null;
    private String address=null;
    private String emailAddress=null;

    public void setName(String name){
        this.name=name;
    }

    public String getName(){
        return this.name;
    }

    public void setAddress(String address){
        this.address=address;
    }

    public String getAddress(){
        return this.address;
    }

    public void setEmailAddress(String emailAddress){
        this.emailAddress=emailAddress;
    }

    public String getEmailAddress(){
        return this.emailAddress;
    }

    /**
     * Reset all properties to their default values.      *
     * @param mapping The mapping used to select this instance
     * @param request The servlet request we are processing      */
    public void reset(ActionMapping mapping, HttpServletRequest request) {
```

```

this.name=null;
this.address=null;
this.emailAddress=null;
}

/***
 * Reset all properties to their default values.
 *  * @param mapping The mapping used to select this instance
 *  * @param request The servlet request we are processing
 *  * @return errors
 */
public ActionErrors validate(
    ActionMapping mapping, HttpServletRequest request ) {
    ActionErrors errors = new ActionErrors();
    if( getName() == null || getName().length() < 1 ) {
        errors.add("name",new ActionMessage("error.name.required"));
    }
    if( getAddress() == null || getAddress().length() < 1 ) {
        errors.add("address",new ActionMessage("error.address.required"));
    }
    if( getEmailAddress() == null || getEmailAddress().length() < 1 ) {
        errors.add("emailaddress",new ActionMessage("error.emailaddress.required"));
    }
    return errors;
}
}

```

The above class populates the Address Form data and validates it. The validate() method is used to validate the inputs. If any or all of the fields on the form are blank, error messages are added to the ActionMapping object. Note that we are using ActionMessage class, ActionError is now deprecated and will be removed in next version.

Now we will create the Action class which is the model part of the application. Our action class simply forwards the request the Success.jsp. Here is the code of the AddressAction class:

AddressAction.java

```

package hpes.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

```

```

public class AddressAction extends Action
{
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception{
        return mapping.findForward("success");
    }
}

```

Now we have to create an entry for the form bean in the struts-config.xml. Add the following lines in the struts-config.xml file:

```
<form-bean name="AddressForm" type="hpes.beans.AddressForm"/>
```

Add the following line in the struts-config.xml file for handling the action "/Address.do":

```

<action path="/Address" type="hpes.actions.AddressAction" name="AddressForm"
    scope="request"
    validate="true"
    input="/pages/Address.jsp">
    <forward name="success" path="/pages/success.jsp"/>
</action>

```

Now create Address.jsp, which is our form for entering the address details. Code for Address.jsp is as follows:

Address.jsp

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html locale="true">

<head>
<title><bean:message key="welcome.title"/></title>
<html:base/>
</head>
<body bgcolor="white">
<html:form action="/Address">
<html:errors/>
<table>
    <tr>
        <td align="center" colspan="2">
            <font size="4">Please Enter the Following Details</font>
        </td>
    </tr>
    <tr>

```

```

<td align="right">
  Name
</td>
<td align="left"> <html:text property="name" size="30" maxlength="30"/> </td>
</tr>
<tr>
  <td align="right">
    Address
  </td>
  <td align="left">
    <html:text property="address" size="30" maxlength="30"/>
  </td>
</tr>

<tr>
  <td align="right">
    E-mail address
  </td>
  <td align="left">
    <html:text property="emailAddress" size="30" maxlength="30"/>
  </td>
</tr>
<tr>
  <td align="right">
    <html:submit>Save</html:submit>
  </td>
  <td align="left">
    <html:cancel>Cancel</html:cancel>
  </td>
</tr>
</table>
</html:form>
</body>
</html:html>

```

User enters the values in the form and clicks the submit form. Form validation is done on the server side and error message is displayed on the jsp page. To display the error on the jsp page `<html:errors/>` tag is used. The `<html:errors/>` tag displays all the errors in one go. To create text box `<html:text .../>` is used in jsp page.

e.g.

```
<html:text property="address" size="30" maxlength="30"/>
```

Above tag creates text box for entering the address. The address is retrieved from and later stored in the property named address in the form-bean.

The tag <html:submit>Save</html:submit> creates the submit button and the tag <html:cancel>Cancel</html:cancel> is used to create the Cancel button.

Add the following line in the index.jsp to create a link for testing the Address.jsp form:

```
<html:link page="/pages/Address.jsp">Test the Address Form</html:link>
```

Build the application and click on the ‘Test the Address Form link’ on the index page to test the newly created screen. You should see the following:



In this section, you learned how to create data entry form using struts, validate and finally send process the business logic in the model part of the struts.

Struts HTML Tags

Struts provides HTML tag library for easy creation of user interfaces. In this section, We'll see what all Struts HTML Tags are available to the JSP for the development of user interfaces.

To use the Struts HTML Tags we have to include the following line in our JSP file:

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
```

above code makes the tags available to the jsp.

Struts HTML Tags

Tag	Description
<html:message key="thekey"/>	Looks up the message corresponding to the given key in the message resources and displays it.
<html:password property="prop" size="10"/>	Tag creates the password field. The string is stored in the property named prop in the form bean.
<html:text property="text1" size="5"/>	Tag creates the text field. The string is retrieved from and later stored in the property named text1 in the form bean.
<html:submit>Submit</html:submit>	Tag creates a submit button with the provided content as the button text.
<html:reset>Reset</html:reset>	Tag creates a reset button with the provided content as the button text.
<html:errors/>	Tag prints all the available error on the page.
<html:file property="fileSelectionBox"/>	Tag creates the file upload element on the form. The property must be of the type org.apache.struts.upload.FormFile.
<html:checkbox property="myCheckBox"/>	Tag creates check box on the form.
<html:hidden property="hiddenfield"/>	Tag creates the hidden html element on the form.
<html:radio value="abc" property="myCheckBox"/>	Tag creates the check box on the form.
<html:select multiple="true" property="selectBox">	Tag creates list box on the form. The property selectBox must be an array of supported data-types, and the user may select several entries. Use <html:options> to specify the entries.
<html:textarea property="myTextArea" value="Hello Struts" />	Tag creates the text area on the form.
<html:form action="/Address" method="post">	Tag is used to create the HTML Form for posting the data on the server.
<html:base/>	Tag generates the base tag. <BASE ...> tells the browser to pretend that the current page is located at some URL other than where the browser found it. Any relative reference will be calculated from the URL given by <BASE HREF="..."> instead of the actual URL. <BASE ...> goes in the <HEAD> section.
<html:html>	Tag renders an HTML <html> Element.

Struts Validator Framework

This section introduces you to the Struts Validator Framework. In this section, you will learn how to use Struts Validator Framework to validate the user inputs on the client browser.

Introduction to Validator Framework

Struts Framework provides the functionality to validate the form data. It can be used to validate the data on the user's browser as well as on the server side. Struts Framework emits the java scripts and it can be used to validate the form data on the client browser. Server side validation of the form can be accomplished by subclassing your FormBean with DynaValidatorForm class.

The Validator framework was developed by David Winterfeldt as third-party add-on to Struts. Now the Validator framework is a part of Jakarta Commons project and it can be used with or without Struts. The Validator framework comes integrated with the Struts Framework and can be used without doing any extra settings.

Using Validator Framework

Validator uses an XML file to pickup the validation rules to be applied to a form. Validation requirements applied to a form are defined in XML. In case, we need special validation rules not provided by the validator framework, we can plug in our own custom validations into the Validator.

The Validator Framework uses two XML configuration files: validator-rules.xml and validation.xml. The validator-rules.xml defines the standard validation routines, these are reusable and used in validation.xml. to define the form specific validations. The validation.xml defines the validations applied to a form bean.

Structure of validator-rules.xml

The validator-rules.xml is provided with the Validator Framework and it declares and assigns the logical names to the validation routines. It also contains the client-side javascript code for each validation routine. The validation routines are java methods plugged into the system to perform specific validations.

Following table contains the details of the elements in this file:

Element	Attributes & Description
form-validation	This is the root node. It contains nested elements for all of the other configuration settings.
global	The validator details specified within this, are global and are accessed by all forms.
validator	<p>The validator element defines what validators objects can be used with the fields referenced by the formset elements.</p> <p>The attributes are:</p> <p>name: Contains a logical name for the validation routine</p> <p>classname: Name of the Form Bean class that extends the subclass of ActionForm class</p> <p>method: Name of the method of the Form Bean class</p> <p>methodParams: parameters passed to the method</p> <p>msg: Validator uses Struts' Resource Bundle mechanism for externalizing</p>

	<p>error messages. Instead of having hard-coded error messages in the framework, Validator allows you to specify a key to a message in the ApplicationResources.properties file that should be returned if a validation fails. Each validation routine in the validator-rules.xml file specifies an error message key as value for this attribute.</p> <p>depends: If validation is required, the value here is specified as 'required' for this attribute.</p> <p>jsFunctionName: Name of the javascript function is specified here.</p>
javascript	<p>Contains the code of the javascript function used for client-side validation. Starting in Struts 1.2.0 the default javascript definitions have been consolidated to commons-validator. The default can be overridden by supplying a <javascript> element with a CDATA section, just as in struts 1.1.</p>

The Validator plug-in (validator-rules.xml) is supplied with a predefined set of commonly used validation rules such as Required, Minimum Length, Maximum length, Date Validation, Email Address validation and more. This basic set of rules can also be extended with custom validators if required.

Structure of validation.xml

This validation.xml configuration file defines which validation routines are to be used to validate Form Beans. You can define validation logic for any number of Form Beans in this configuration file. In that definition, you specify the validations you want to apply to the Form Bean's fields. The definitions in this file use the logical names of Form Beans from the struts-config.xml file along with the logical names of validation routines from the validator-rules.xml file to tie the two together.

Element	Attributes & Description
form-validation	This is the root node. It contains nested elements for all of the other configuration settings
global	The constant details are specified in <constant> element within this element.
constant	Constant properties are specified within this element for pattern matching.
constant-name	Name of the constant property is specified here
constant-value	Value of the constant property is specified here.
formset	This element contains multiple <form> elements
form	<p>This element contains the form details. The attributes are: name: Contains the form name. Validator uses this logical name to map the validations to a Form Bean defined in the struts-config.xml file</p>
field	<p>This element is inside the form element, and it defines the validations to apply to specified Form Bean fields.</p> <p>The attributes are: property: Contains the name of a field in the specified Form Bean depends: Specifies the logical names of validation routines from the validator-rules.xml file that should be applied to the field.</p>

arg	A key for the error message to be thrown incase the validation fails, is specified here
var	Contains the variable names and their values as nested elements within this element.
var-name	The name of the criteria against which a field is validated is specified here as a variable
var-value	The value of the field is specified here

Example of form in the validation.xml file:

```
<!-- An example form -->
<form name="logonForm">
<field property="username" depends="required">
    <arg key="logonForm.username"/>
</field>
<field property="password" depends="required,mask">
    <arg key="logonForm.password"/>
    <var>
        <var-name>mask</var-name>
        <var-value>^/[0-9a-zA-Z]*$</var-value>
    </var>
</field>
</form>
```

The `<html:javascript>` tag allows front-end validation based on the xml in validation.xml. For example the code:

```
<html:javascript formName="logonForm" dynamicJavascript="true" staticJavascript="true" />
```

generates the client side java script for the form "logonForm" as defined in the validation.xml file. The `<html:javascript>` when added in the jsp file generates the client site validation script.

In the next section, we will create a new form for entering the address and enable the client side java script with the Validator Framework.

Client Side Address Validation in Struts

In this section, we will create JSP page for entering the address and use the functionality provided by Validator Framework to validate the user data on the browser. Validator Framework emits the JavaScript code which validates the user input on the browser. To accomplish this we have to follow the following steps:

- Enabling the Validator plug-in: This makes the Validator available to the system.
- Create Message Resources for the displaying the error message to the user.
- Developing the Validation rules: We have to define the validation rules in the validation.xml for the address form. Struts Validator Framework uses this rule for generating the JavaScript for validation.

- Applying the rules: We are required to add the appropriate tag to the JSP for generation of JavaScript.
- Build and test: We are required to build the application once the above steps are done before testing.

Enabling the Validator plug-in

To enable the validator plug-in open the file struts-config.xml and make sure that following line is present in the file.

```
<!-- Validator plugin -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

Creating Message Resources

Message resources are used by the Validator Framework to generate the validation error messages. In our application we need to define the messages for name, Address and E-mail address. Open the Struts\strutstutorial\web\WEB-INF\MessageResources.properties file and add the following lines:

```
AddressForm.name=Name
AddressForm.address=Address
AddressForm.emailAddress=E-mail address
```

Developing Validation rules

In this application we are adding only one validation that the fields on the form should not be blank. Add the following code in the validation.xml.

```
<!-- Address form Validation-->
<form name="AddressForm">
    <field property="name" depends="required">
        <arg key="AddressForm.name"/>
    </field>
    <field property="address" depends="required">
        <arg key="AddressForm.address"/>
    </field>
    <field property="emailAddress" depends="required">
        <arg key="AddressForm.emailAddress"/>
    </field>
</form>
```

The above definition defines the validation for the form fields name, address and emailAddress. The attribute depends="required" instructs the Validator Framework to generate the JavaScript that checks that the fields are not left blank. If the fields are left blank then JavaScript shows the error message. In the error message the message are taken from the key defined in the <arg key=".."/> tag. The value of key is taken from the message resources (Struts\strutstutorial\web\WEB-INF\MessageResources.properties).

Applying Validation rules to JSP

Now create the AddressJavascriptValidation.jsp file to test the application. The code for AddressJavascriptValidation.jsp is given below:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html locale="true">
<head>
    <title><bean:message key="welcome.title"/></title> <html:base/>
</head>
<body bgcolor="white">
<html:form action="/AddressJavascriptValidation" method="post" onsubmit="return
validateAddressForm(this);">

<div align="left">
<p>
This application shows the use of Struts Validator.<br>
The following form contains fields that are processed by Struts Validator.<br>
Fill in the form and see how JavaScript generated by Validator Framework validates the form.
</p>

<p>
<html:errors/>
</p>
<table>

<tr>
<td align="center" colspan="2">
<font size="4"><b>Please Enter the Following Details</b></font> </td>
<tr>
<td align="right">
<b>Name</b>
</td>
<td align="left">
<html:text property="name" size="30" maxlength="30"/> </td>
</tr>
```

```

<tr>
<td align="right">
<b>Address</b>
</td>
<td align="left">
<html:text property="address" size="30" maxlength="30"/> </td>
</tr>
<tr>
<td align="right">
<b>E-mail address</b>
</td>
<td align="left">
<html:text property="emailAddress" size="30" maxlength="30"/> </td>
</tr>
<tr>
<td align="right">
<html:submit>Save</html:submit>
</td>
<td align="left">
<html:cancel>Cancel</html:cancel>
</td>
</tr>
</table>
</div>

<!-- Begin Validator Javascript Function--> <html:javascript formName="AddressForm"/> <!--
End of Validator Javascript Function-->
</html:form>
</body>
</html:html>

```

The code <html:javascript formName="AddressForm"/> is used to plug-in the Validator JavaScript.

Create the following entry in the struts-config.xml for the mapping the AddressJavascript Validation url for handling the form submission through AddressJavascriptValidation.jsp.

```

<action
    path="/AddressJavascriptValidation"      type="hpes.actions.AddressAction"
    name="AddressForm"                      scope="request"
    validate="true"
    input="/pages/AddressJavascriptValidation.jsp">
    <forward name="success" path="/pages/success.jsp"/>
</action>

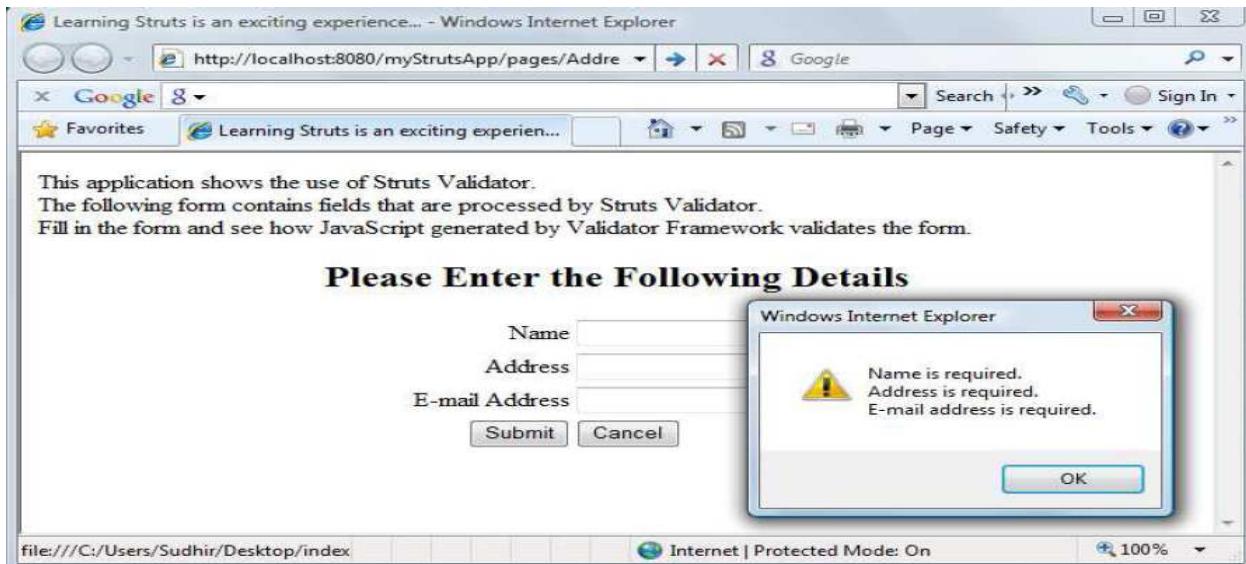
```

Add the following line in the index.jsp to call the form.

```
<li>
<html:link page="/pages/AddressJavascriptValidation.jsp">
    Client Side Validation for Address Form
</html:link>
<br>
The Address Form that validates the data on the client side using Stuts Validator generated
JavaScript.
</li>
```

Building Example and Testing

Build & deploy the application on the server. Open the browser and navigate to the AddressJavascriptValidation.jsp page. Your browser should show the following output.



If the fields are left blank and Save button is clicked, browser shows the error message.

In this section, you learned how to use Struts Validator Framework to validate the form on client browser.

Creating Custom Validators in Struts

In this tutorial you will learn how to develop Custom Validators in your Struts 1.3 applications. Struts Validator framework provides many validation rules that can be used in the web applications. If your application needs special kind of validation, then you can extend the validator framework to develop your own validation rule.

The client-side validation in Struts is well known. Here are some of the available features:

required	requiredif	validwhen	minlength
maxlength	mask	byte	short
integer	long	float	double
byteLocale	shortLocale	integerLocale	longLocale
floatLocale	doubleLocale	date	intRange
longRange	floatRange	doubleRange	creditCard
email	url		

These are found in the validator-rules.xml inside the <validator> tags. The validator-rules.xml file is found in the commons-validator jar.

To use our matchname validator create a file validation.xml and add the following lines:

```
<!-- Name form Validation-->
<form-validation>
<formset>
<form name="AdminForm">
<field property="name" depends="matchname">
    <arg0 key="AddressForm.name"/>
</field>
</form>
</formset>
</form-validation>
```

Copy the files validation.xml and validator-rules.xml to the directory where your struts-config.xml resides. Let us say it is WEB-INF. Next we have to create the error message for errors.name. Create a directory WEB-INF/resources and a file in this directory with the name application.properties. Add the following lines to application.properties

```
AdminForm.name=Name
errors.name={0} should be administrator.
errors.required={0} is required.
errors.minLength={0} can not be less than {1} characters. errors.maxLength={0} can not be greater
than {1} characters.

errors.invalid={0} is invalid.
errors.byte={0} must be a byte.
errors.short={0} must be a short.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.float={0} must be a float.
errors.double={0} must be a double.
```

*errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}. errors.creditcard={0} is an invalid credit card number. errors.email={0} is an invalid e-mail address.*

Edit struts-configuration.xml and add the following lines

```
<form-bean name="AdminForm" type="test.AdminForm"/>
<action
    path="/AdminFormValidation"
    type="test.AdminAction"
    name="AdminForm"
    scope="request"
    validate="true"
    input="admin.jsp">
    <forward name="success" path="success.jsp"/>
</action>

<message-resources parameter="resources/application"/>
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
        property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

Create a JSP file as follows:

```
<%@ taglib uri="struts-bean.tld" prefix="bean" %>
<%@ taglib uri="struts-html.tld" prefix="html" %>
<html:html>
<head>
    <title>Administrator Test</title>
    <html:base/>
</head>
<body bgcolor="white">
<html:form action="/AdminFormValidation" method="post" onsubmit="return
validateAdminForm(this);">

<div align="left">

<p>
This application shows the use of Struts Validator.<br> The following form contains fields that are
processed by Struts Validator.<br> Fill in the form and see how JavaScript generated by Validator
Framework validates the form.
</p>
```

```

<p>
<html:errors/>
</p>
<table>
<tr>
<td align="right">
<b>Name</b>
</td>
<td align="left">
<html:text property="name" size="30" maxlength="30"/> </td>
</tr>
<tr>
<td align="right">
<html:submit>Save</html:submit>
</td>
<td align="left">
<html:cancel>Cancel</html:cancel>
</td>
</tr>
</table>
</div>

<!-- Begin Validator Javascript Function-->
<html:javascript formName="AddressForm"/> <!-- End of Validator Javascript Function-->

</html:form>
</body>
</html:html>

```

Then we create the success.jsp

```
<% out.println("SUCCESS") %>
```

Then we create the Java Class for the AdminForm

```

package test;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

/**
 * Form bean for the Admin Entry Screen. * */
public class AdminForm extends ActionForm {
    private String name=null;

```

```

public void setName(String name){
    this.name=name;
}

public String getName(){
    return this.name;
}

/***
 * Reset all properties to their default values.      *
 * @param mapping The mapping used to select this      *
 * @param request The servlet request we are processing      */
public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.name=null;
}

/***
 * Reset all properties to their default values.      *
 * @param mapping The mapping used to select this instance      *
 * @param request The servlet request we are processing      * @return errors      */
public ActionErrors validate(
    ActionMapping mapping, HttpServletRequest request ) {
    ActionErrors errors = new ActionErrors();
    if( getName() == null || getName().length() < 1 ) {
        errors.add("name",new ActionMessage("error.name.required"));
    }
    return errors;
}
}

```

Create the AdminAction.java

```

package test;
import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward; import org.apache.struts.action.ActionMapping;
public class AdminAction extends Action
{
    public ActionForward execute(ActionMapping mapping,ActionForm form,    HttpServletRequest
request, HttpServletResponse response) throws Exception{
        return mapping.findForward("success");
    }
}

```

Finally compile the classes and restart the web server and view the AdminForm.jsp

Developing Simple Struts Tiles Application

In this section, we'll see how to develop simple Struts Tiles Application. You will learn how to setup the Struts Tiles and create an example page with it.

What is Struts Tiles?

Tiles is a framework for the development of user interface. Tiles enables the developers to develop the web applications by assembling the reusable tiles (jsp, html, etc..). Tiles uses the concept of reuse and enables the developers to define a template for the web site and then use this layout to populate the content of the web site. For example, if you have to develop a web site having more than 500 page of static content and many dynamically generated pages. The layout of the web site often changes according to the business requirement. In this case you can use the Tiles framework to design the template for the web site and use this template to populate the contents. In future, if there is any requirement of site layout change, then you have to change the layout in one page. This will change the layout of the whole web site.

Steps To Create Tiles Application

Tiles is a very useful framework for the development of web applications. Here are the steps necessary for adding Tiles to your Struts application:

- Add the Tiles Tag Library Descriptor (TLD) file to the web.xml.
- Create layout JSPs.
- Develop the web pages using layouts.
- Repackage, run and test application.

Add the Tiles TLD to web.xml file

Following entry is required in the web.xml file before you can use the tiles tags in your application.

```
<taglib>
  <taglib-uri>/tags/struts-tiles</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
```

Create layout JSPs.

Our web application layout is divided into four parts: Banner, Left Navigation Bar, Content Area and Bottom of the page for copy right information. Here is the code for our template (template.jsp):

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<html>
<head>
```

```

<title><tiles:getAsString name="title" ignore="true"/></title>
</head>
<body>
<table border="1" cellpadding="0" cellspacing="0" width="100%" bordercolor="#000000"
bgcolor="#E7FDDE"> <tr>
<td width="100%" colspan="2" valign="top"><tiles:insert attribute="header"/></td>
</tr>
<tr>
<td width="23%"><tiles:insert attribute="menu"/></td> <td width="77%" valign="top"
valign="top"><tiles:insert attribute="body"/></td> </tr>
<tr>
<td width="100%" colspan="2" valign="top">
<tiles:insert attribute="bottom"/>
</td>
</tr>
</table>
</body>
</html>

```

We have defined the structure for web application using the appropriate html and did the following things:

- Referenced the /WEB-INF/struts-tiles.tld TLD.
- Used the string parameters to display title using the tiles:getAsString tag. If the attribute ignore="true" then Tiles ignore the missing parameter. If this is true then the Tiles framework will throw an exception in case the parameter is missing.
- To insert the content JSP, the tiles:insert tag is used, which inserts any page or web resources that framework refers to as a title. For Example <tiles:insert attribute="header"/> inserts the header web page.

Develop the web pages using layouts

Now we will use tile layout to create the content page in the application. For every content page there is additional jsp file for inserting the content in the Layout, so we have to create two jsp files one for content and another for displaying the content. In our example these file are example.jsp and content.jsp. Here is the code for both the files:

content.jsp

```

<p align="left"><font color="#000080" size="5">Using Tiles is very easy.</font></p>
<p align="left"><font color="#000080" size="5">This is the content page</font></p>
The content.jsp simply define the content of the page. The content may be dynamic or static
depending on the requirements.

```

example.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert page="/tiles/template.jsp" flush="true">
<tiles:put name="title" type="string" value="Welcome" />
<tiles:put name="header" value="/tiles/top.jsp" />
<tiles:put name="menu" value="/tiles/left.jsp" />
<tiles:put name="body" value="/tiles/content.jsp" />
<tiles:put name="bottom" value="/tiles/bottom.jsp" />
</tiles:insert>
```

The code `<tiles:insert page="/tiles/template.jsp" flush="true">` specifies the tiles layout page to be used. We have set the flush attribute to true, this makes the tile file to be written to browser before the rest of the page. To specify the title of the page `<tiles:put name="title" type="string" value="Welcome" />` is used. The following code is used to insert the actual pages in the template.:

```
<tiles:put name="header" value="/tiles/top.jsp" />
<tiles:put name="menu" value="/tiles/left.jsp" />
<tiles:put name="body" value="/tiles/content.jsp" />
<tiles:put name="bottom" value="/tiles/bottom.jsp" />
```

The top.jsp will be inserted in the layout's header region. The left.jsp will be inserted in the layout's menu region. The content.jsp wil be inserted in the layout's body region and the bottom.jsp will be inserted in the bottom region.

Repackage, run and test application

Add the following code in the index.jsp to test the this tile example:

```
<li>
<html:link page="/tiles/example.jsp">Tiles Example</html:link> <br>
Example of creating first tile application.
</li>
```

Build the application and deploy on the server. To test the application go to the index.jps and click on the Tiles Example link.

Using tiles-defs.xml in Tiles Application

In the last section, we learned how to forward the request (call) to a jsp page which specifies which tiles layout definition should be used to apply to the content. In this section, we'll see how to use a definition in the tiles-defs.xml for generating the content.

We can define the definition in the tiles-defs.xml which specifies the different components to "plugin" to generate the output. This eliminates the need to define extra jsp file for each content file.

For example, in the last section, we defined example.jsp to display the content of content.jsp file. In this section, I will show you how to eliminate the need of extra jsp file using tiles-defs.xml file.

Steps to Use the tiles-defs.xml

- Setup the Tiles plugin in struts-config.xml file.

Add the following code in the struts-config.xml (If not present). This enables the TilesPlugin to use the /WEB-INF/tiles-defs.xml file.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
<!-- Path to XML definition file -->
<set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
<!-- Set Module-awareness to true -->
<set-property property="moduleAware" value="true" />
</plug-in>
```

Defining the tiles-defs.xml

In this file we are defining the different components to "plugin". Here is the code:

```
<definition name="Tiles.Example" page="/tiles/template.jsp">
<put name="title" type="string" value="Welcome" />
<put name="header" value="/tiles/top.jsp" />
<put name="menu" value="/tiles/left.jsp" />
<put name="body" value="/tiles/content.jsp" />
<put name="bottom" value="/tiles/bottom.jsp" />
</definition>
```

The name of the definition is Tiles.Example, we will use this in struts-config.xml (While creating forwards in struts-config.xml file). The page attribute defines the template file to be used and the put tag specifies the different components to "plugin". Your tiles-defs.xml should look like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
<definition name="Tiles.Example" page="/tiles/template.jsp">
<put name="title" type="string" value="Welcome" />

<put name="header" value="/tiles/top.jsp" />
<put name="menu" value="/tiles/left.jsp" />
<put name="body" value="/tiles/content.jsp" />
<put name="bottom" value="/tiles/bottom.jsp" />
```

```
</definition>
<definition name="${YOUR_DEFINITION_HERE}"> </definition>
</tiles-definitions>
```

Configure the Struts Action to use Tiles Definition

Open the struts-config.xml file and add the following code:

```
<action path="/Tiles/Example" forward="Tiles.Example"/>
```

With Tiles, the action points to the Tiles definition, as shown in the above code. In this code, we are using the Tiles.Example definition which we have defined in the tiles-defs.xml file. Without Tiles, forward and action definitions point directly to JSPs. With Tiles, they point to the page's definition in the Tiles configuration file.

Testing the Application

Create a link in index.jsp to call the Example. Code to do this is given below.

```
<li>
<html:link page="/Tiles/Example.do">Using tiles-defs.xml</html:link> <br>
Example shows you how to use tiles-defs.xml file.
</li>
```

Build and deploy the application on the server. Type <http://localhost:8080/myStrutsApp/index.jsp> in the browser and select the Using tiles-defs.xml link. Your browser should show the page.

Struts DynaActionForm

In this section, you will learn how to create Struts DynaActionForm. We will recreate our address form with Struts DynaActionForm. DynaActionForm is specialized subclass of ActionForm that allows the creation of form beans with dynamic sets of properties, without requiring the developer to create a Java class for each type of form bean. DynaActionForm eliminates the need of FormBean class and now the form bean definition can be written into the struts-config.xml file. So, it makes the FormBean declarative and this helps the programmer to reduce the development time.

In this section, we will recreate the add form with the help of DynaActionForm. It also shows you how you can validate user input in the action class.

Adding DynaActionForm Entry in struts-config.xml

First, we will add the necessary entry in the struts-config.xml file. Add the following entry in the struts-config.xml file. The form bean is of org.apache.struts.action.DynaActionForm type. The `<form-property/>` tag is used to define the property for the form-bean.

We have defined three properties for our dynamic form bean.

```

<form-bean name="DynaAddressForm" type="org.apache.struts.action.DynaActionForm">
<form-property name="name" type="java.lang.String"/>
<form-property name="address" type="java.lang.String"/>
<form-property name="email" type="java.lang.String" />
</form-bean>

```

Adding action mapping

Add the following action mapping in the struts-config.xml file:

```

<action path="/DynaAddress" type="hpes.actions.AddressDynaAction"
name="DynaAddressForm"
scope="request"
validate="true"
input="/pages/DynaAddress.jsp">

<forward name="success" path="/pages/success.jsp"/>
<forward name="invalid" path="/pages/DynaAddress.jsp" />
</action>

```

Creating Action Class: Code for action class is as follows:

```

package hpes.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.DynaActionForm;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.action.ActionMessage;
public class AddressDynaAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception{

```

```

        DynaActionForm addressForm = (DynaActionForm)form;
        //Create object of ActionMesssages
        ActionMessages errors = new ActionMessages();    //Check and collect errors
        if(((String)addressForm.get("name")).equals("")) {
            errors.add("name",new ActionMessage("error.name.required"));
        }

        if(((String)addressForm.get("address")).equals("")) {

```

```

errors.add("address",new ActionMessage("error.address.required"));      }

if((String)addressForm.get("email").equals("")) {
    errors.add("email",new ActionMessage("error.emailaddress.required"));      }

//Saves the error
saveErrors(request,errors);
//Forward the page
if(errors.isEmpty()){
    return mapping.findForward("success");
} else{
    return mapping.findForward("invalid");
}
}
}
}

```

Creating the JSP file

We will use the Dyna Form DynaAddressForm created above in the jsp file. Here is the code of the jsp(DynaAddress.jsp) file.

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html locale="true">
<head>
<title><bean:message key="welcome.title"/></title>
</head>
<body bgcolor="white">
<html:form action="/DynaAddress" method="post"> <table>
<tr>
<td align="center" colspan="2">
<font size="4">Please Enter the Following Details</font> </tr>
<tr>
<td align="left" colspan="2">
<font color="red"><html:errors/></font>

</tr>
<tr>
<td align="right">
Name
</td>
<td align="left">

```

```

<html:text property="name" size="30" maxlength="30"/> </td>
</tr>
<tr>
<td align="right">Address</td>
<td align="left"><html:text property="address" size="30" maxlength="30"/>
</td>
</tr>
<tr>
<td align="right">
E-mail address
</td>
<td align="left">
<html:text property="email" size="30" maxlength="30"/> </td>
</tr>
<tr>
<td align="right">
<html:submit>Save</html:submit>
</td>
<td align="left">
<html:cancel>Cancel</html:cancel>
</td>
</tr>
</table>
</html:form>
</body>
</html:html>

```

Add the following line in the index.jsp to call the form.

```

<li>
<html:link page="/pages/DynaAddress.jsp">Dyna Action Form Example</html:link>
<br>
Example shows you how to use DynaActionForm.
</li>

```

Building Example and Testing

Build & deploy the application on the server. Open the browser and navigate to the DynaAddress.jsp page. See the result.

Struts File Upload Example

In this section, you will learn how to use Struts to write a program to upload files. The interface org.apache.struts.upload.FormFile is the heart of the struts file upload application. This interface represents a file that has been uploaded by a client. It is the only class in upload package which is typically referenced directly by a Struts application.

Creating Form Bean

Our form bean class contains only one property theFile, which is of type org.apache.struts.upload.FormFile.

Here is the code of FormBean (StrutsUploadForm.java):

```
package hpes.beans;
import org.apache.struts.action.*;
import org.apache.struts.upload.FormFile;

/**
 * Form bean for Struts File Upload. * */
public class StrutsUploadForm extends ActionForm {
    private FormFile theFile;

    public FormFile getTheFile() {
        return theFile;
    }
    public void setTheFile(FormFile theFile) {
        this.theFile = theFile;
    }
}
```

Creating Action Class

Our action class simply calls the getFile() function on the FormBean object to retrieve the reference of the uploaded file. Then the reference of the FormFile is used to get the uploaded file and its information. Here is the code of our action class(StrutsUploadAction.java):

```
package hpes.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
```

```

import org.apache.struts.upload.FormFile;
<太后
 * Struts File Upload Action Form.
 *
 */
public class StrutsUploadAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception{
        StrutsUploadForm myForm = (StrutsUploadForm)form;
        // Process the FormFile
        FormFile myFile = myForm.getFile();
        String contentType = myFile.getContentType();
        String fileName = myFile.getFileName();
        int fileSize = myFile.getFileSize();
        byte[] fileData = myFile.getFileData();
        System.out.println("contentType: " + contentType);
        System.out.println("File Name: " + fileName);
        System.out.println("File Size: " + fileSize);
        return mapping.findForward("success");
    }
}

```

Defining form Bean in struts-config.xml file

Add the following entry in the struts-config.xml file for defining the form bean:

```
<form-bean name="FileUpload" type="hpes.beans.StrutsUploadForm"/>
```

Defining Action Mapping

Add the following action mapping entry in the struts-config.xml file:

```

<action
path="/FileUpload"
type="hpes.actions.StrutsUploadAction"
name="FileUpload"
scope="request"
validate="true"
input="/pages/FileUpload.jsp">
    <forward name="success" path="/pages/uploadsuccess.jsp"/>
</action>

```

Developing jsp page

Code of the jsp (FileUpload.jsp) file to upload is as follows:

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>

<html:html locale="true">
<head>
<title>Struts File Upload Example</title>
<html:base/>
</head>
<body bgcolor="white">
<html:form action="/FileUpload" method="post" enctype="multipart/form-data">
<table>
<tr>
<td align="center" colspan="2">
<font size="4">Please Enter the Following Details</font> </tr>
<tr>
<td align="left" colspan="2">
<font color="red"><html:errors/></font>
</tr>

<tr>
<td align="right">File Name</td>
<td align="left"><html:file property="theFile"/></td> </tr>
<tr>
<td align="center" colspan="2">
<html:submit>Upload File</html:submit>
</td>
</tr>
</table>
</html:form>
</body>
</html:html>

```

Note that we are setting the encrypt property of the form to enctype="multipart/form-data".
Code for the success page (uploadsuccess.jsp) is as follows:

```

<html>
<head>
<title>Success</title>
</head>
<body>
<p align="center"><font size="5" color="#000080">File Successfully Received</font></p>
</body>

```

```
</html>
```

Add the following line in the index.jsp to call the form.

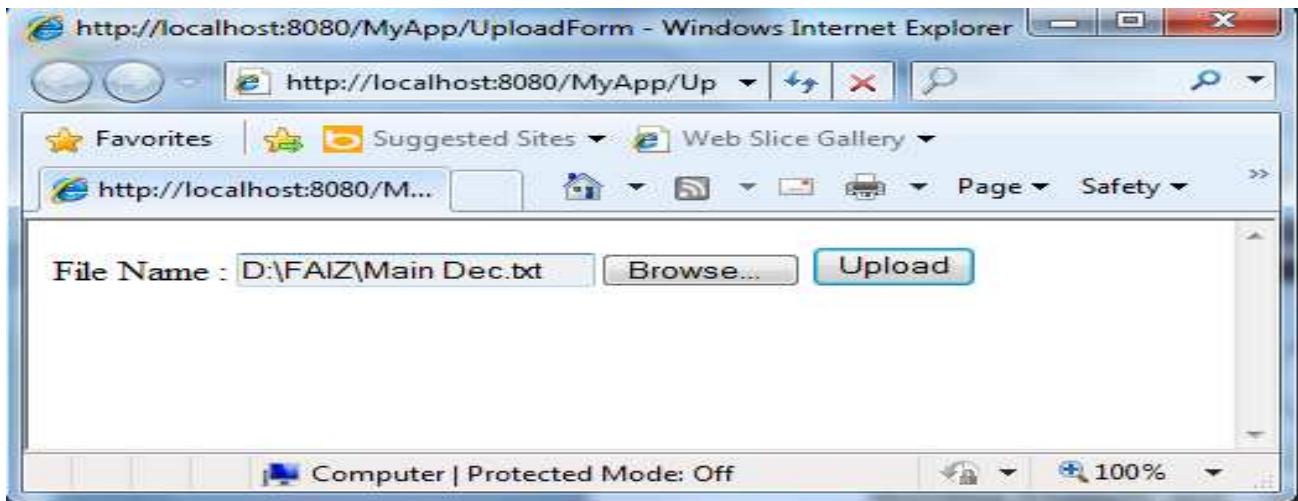
```
<li>
```

```
<html:link page="/pages/FileUpload.jsp">Struts File Upload</html:link> <br>
```

Example shows you how to Upload File with Struts.

Building Example and Testing

Build & deploy the application. Open the browser and navigate to the FileUpload.jsp page. Your browser should display the file upload form:



Struts File Upload and Save

The above section didn't contain any code illustrating how to save the file on the server . Now, the current example will provide you with the code to upload the file ,in the upload directory of server.

In this section, you'll learn how to use Struts program to upload a file on the Server and display a link to the user to download the uploaded file . The class org.apache.struts.upload.FormFile has a prime role in uploading a file in a Struts application. This class represents a file that has been uploaded by a client. It is the only class in Upload package which is referenced directly by a Struts application.

Creating Form Bean

Our form bean class contains only one property theFile, which is of type org.apache.struts.upload.FormFile.

Here is the code of FormBean (StrutsUploadAndSaveForm.java):

```
package hpes.beans;
import org.apache.struts.action.*;
/**
 * Form bean for Struts File Upload.  */
public class StrutsUploadAndSaveForm extends ActionForm {
    private FormFile theFile;
    public FormFile getTheFile() {
        return theFile;
    }
    public void setTheFile(FormFile theFile) {
        this.theFile = theFile;
    }
}
```

Creating Action Class

In the previous section entitled "Struts File Upload Example", we just had action class simply calling the getTheFile() function on the FormBean object to retrieve the reference of the uploaded file. Then the reference of the FormFile was used to get the uploaded file and its information. Now further we retrieve the Servers upload directory's real path using ServletContext's getRealPath() and saving the file.

Code of StrutsUploadAndSaveAction.java:

```
package hpes.actions;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.upload.FormFile;
import java.io.*;

/**
 * Struts File Upload Action Form.
 *
 */
public class StrutsUploadAndSaveAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception{
        StrutsUploadAndSaveForm myForm = (StrutsUploadAndSaveForm)form;
```

```

// Process the FormFile
FormFile myFile = myForm.getTheFile();

String contentType = myFile.getContentType();
//Get the file name
String fileName = myFile.getFileName();
int fileSize = myFile.getFileSize();
byte[] fileData = myFile.getFileData();
//Get the servers upload directory real path name
String filePath = getServlet().getServletContext().getRealPath("/") + "upload";
/* Save file on the server */
if(!fileName.equals ""){
    System.out.println("Server path:" + filePath);
//Create file
    File fileToCreate = new File(filePath, fileName);
    //If the file does not exists create file
    if(!fileToCreate.exists()){
        FileOutputStream fileOutStream = new FileOutputStream(fileToCreate);
        fileOutStream.write(myFile.getFileData());
        fileOutStream.flush();
        fileOutStream.close();
    }
}
//Set file name to the request object
request.setAttribute("fileName", fileName);
return mapping.findForward("success");
}
}

```

Defining form Bean in struts-config.xml file

Add the following entry in the struts-config.xml file for defining the form bean:

```
<form-bean name="FileUploadAndSave" type="hpes.beans.StrutsUploadAndSaveForm"/>
```

Defining Action Mapping

Add the following action mapping entry in the struts-config.xml file:

```

<action
path="/FileUploadAndSave" type="hpes.actions.StrutsUploadAndSaveAction"
name="FileUploadAndSave" scope="request" validate="true"
input="/pages/FileUploadAndSave.jsp">
<forward name="success" path="/pages/downloaduploadedfile.jsp"/>
</action>

```

Developing jsp pages

Code of the jsp (FileUploadAndSave.jsp) file to upload is as follows:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html locale="true">
<head>
<title>Struts File Upload and Save Example</title>
</head>
<body bgcolor="white">
<html:form action="/FileUploadAndSave" method="post" enctype="multipart/form-data">
<table>
<tr>
<td align="center" colspan="2">
<font size="4">File Upload on Server</font> </tr>
<tr>
<td align="left" colspan="2">
<font color="red"><html:errors/></font>
</tr>
<tr>
<td align="right">File Name</td>
<td align="left"><html:file property="theFile"/></td> </tr>
<tr>
<td align="center" colspan="2">
<html:submit>Upload File</html:submit>
</td>
</tr>
</table>
</html:form>
</body>
</html:html>
```

Code for the success page (downloaduploadedfile.jsp) is as follows:

```
<html>
<head>
<title>Success</title>
</head>
<body>
<%
String fileName=(String)request.getAttribute("fileName"); %>
<p align="center"><font size="5" color="#000080">File Successfully Received</font></p>
<p align="center"><a href="upload/<%=fileName%>">Click here to download</a></p>
```

```
</body>  
</html>
```

Add the following line in the index.jsp to call the form.

```
<li>  
<html:link page="/pages/FileUploadAndSave.jsp">Struts File Upload</html:link>  
<br>  
Example shows you how to Upload File with Struts.  
</li>
```

Building Example and Testing

Build & deploy the application on the server. Open the browser and navigate to the Struts File Upload page. Your browser should display the file upload form:

